Formalna verifikacija programske potpore: Uvod

Sveučilište u Zagrebu

Fakultet elektrotehnike i računarstva

Pripremio: izv. prof. dr. sc. Alan Jović

Ak. god. 2022./2023.

Što su formalne metode (FM)?

- To su **matematički zasnovane tehnike** za specificiranje zahtjeva i arhitekture u oblikovanju i razvoju, te za verifikaciju sklopovskih i programskih sustava
- Uporaba FM je motivirana istim argumentima kao i u drugim inženjerskim disciplinama: matematički zasnovana analiza **doprinosi pouzdanosti i robusnosti konačnog proizvoda**
- FM nisu zamjena za ispitivanje (testiranje) već su to dva međusobno **komplementarna** skupa tehnika
  - Ispitivanje – dinamička verifikacija i formalna verfikacija (dio FM) se nadopunjuju
  - U FM usredotočenost je češće na **statičko** (simboličko) rasuđivanje o sustavima

Što je zajedničko većini formalnih metoda?

- Jezik
  - FM su matematički način zapisa ili računalni jezik s formalnom semantikom koji omogućuje opis svojstava sustava i svojstava koja se od sustava očekuju. Sustav je opisan kao matematički objekt na rigorozan i analitički način putem nekog jezika
- Alat
  - FM dolaze u obliku računalnih alata koji osiguravaju da se sustav izvede i u konstrukcijskom i u funkcionalnom smislu prema očekivanjima korisnika
- Metodologija
  - Da bi bili učinkovite, FM moraju biti dobro integrirane u industrijsku praksu, stoga većina formalnih metoda ima smjernice za ispravnu uporabu u razvoju stvarnih sustava

Formalne metode u primjeni

- Primjena u životnom ciklusu (engl. *life-cycle*) razvoja sustava
  - u svim fazama ciklusa razvoja
  - samo u nekim fazama razvoja, često u fazi validacije i verifikacije
- Primjena na komponente sustava
  - na sve komponente
  - samo na neke, i to one složenije i kritične (češće)
- Primjena na svojstva (engl. *properties*) komponente
  - primjena na sva funkcijska svojstva (puna funkcionalnost)
  - primjena samo na neka svojstva, obično na ona kritična

Formalne metode u industrijskoj praksi

FM se najčešće koriste za sustave gdje je cijena zatajenja visoka

- **Za sustave s kritično važnim ciljem (engl. *mission-critical system*)**
  - Zatajenja su ovdje posebno skupa, a pogreške teško otklonjive nakon puštanja u pogon; najčešće se verificira sklopovlje
  - Npr. navigacija u raketnoj industriji, vojna oprema, proizvodnja čipova

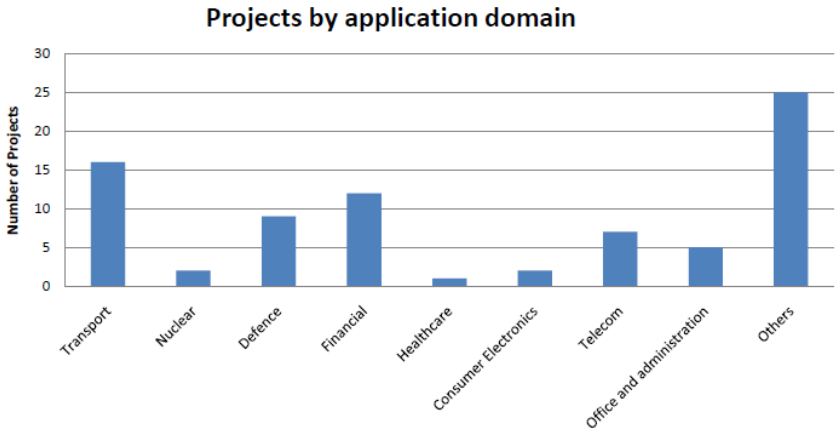
- **Za „po život kritične" sustave (*engl. life-critical system, safety critical system*)**
  - FM su zakonski zahtijevane po tehničkim normama ili certifikacijskim autoritetima; verificira se i programska potpora i sklopovlje
  - Npr. civilno zrakoplovstvo, zdravstvo, željeznice, nuklearne elektrane, roboti

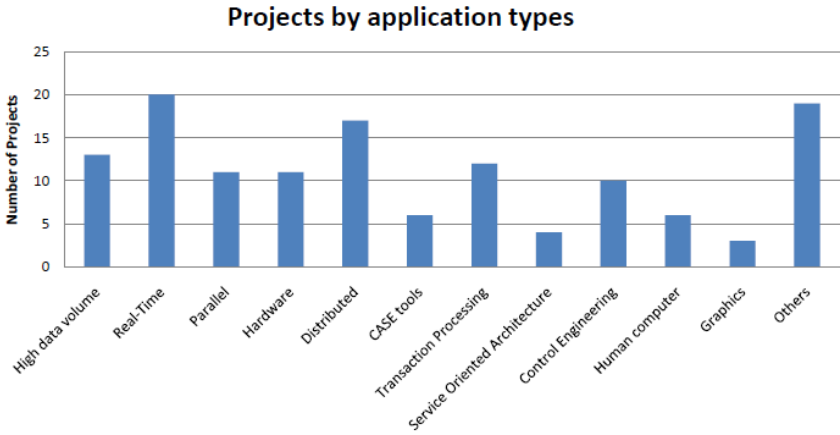
- **Za informacijske i poslovne sustave s visokom razinom sigurnosti (engl. *high-security system*)**
  - Često zahtijevaju rad u skladu s normama i razinama zaštite; verificira se najčešće programska potpora
  - Npr. pristup tajnim državnim podacima, bankovni sustavi

Formalne metode u industrijskoj praksi

- Postoje dokazi da uporaba FM povećava kvalitetu, **smanjuje cijenu i skraćuje vrijeme** stavljanja proizvoda na tržište (engl. *time-to-market*)
- Npr. J. Woodcock, P. G. Larsen, J. Bicarregui, and J. S. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys, 41(4), 2009.*

  62 industrijska projekta:
  - Kvaliteta: 92% poboljšanje, 8% nema učinka
  - Vrijeme: 35% poboljšanje, 53% nema učinka, 12% pogoršanje
  - Cijena: 37% poboljšanje, 56% nema učinka, 7% pogoršanje
  - Većina sudionika u studiji složila se da je korištenje formalnih metoda bilo uspješno (jako suglasni: 61%, suglasni: 34%, miješano mišljenje: 5%).



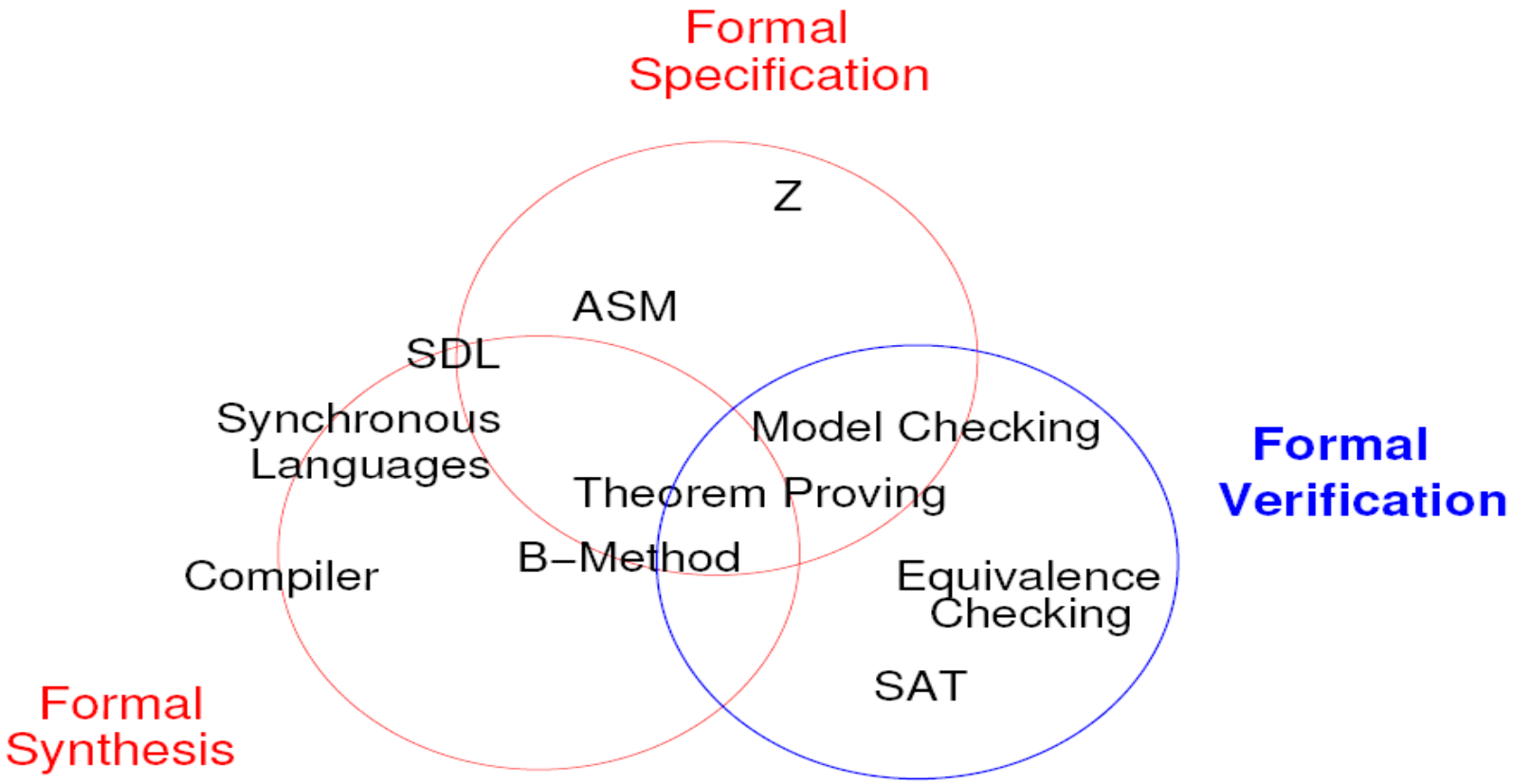


Što se sve može tvrditi o FM?

- Pozitivno:
  - FM su korisne u ranom otkrivanju i eliminaciji pogrešaka
  - FM mogu klijentima bolje pomoći u razumijevanju toga što točno kupuju
  - FM rade tako da vas prisile da puno razmišljate o sustavu koji planirate izgraditi prije kodiranja
  - FM mogu smanjiti cijenu i trajanje razvoja projekta
- Negativno
  - FM u većini slučajeva ne mogu garantirati da je sustav bez pogrešaka – čak i ako je sama FM savršena, čovjek koji ju koristi nije
  - FM su teške za naučiti i primijeniti u praksi
  - Za uspješnu primjenu, često se treba kombinirati više FM

Klasifikacija formalnih metoda

(izvor: Daniel Kroening, University of Oxford, UK)



Formalna specifikacija

- Apstrahira nepotrebne implementacijske detalje.
- To je matematički model sustava koji želimo razviti visoke razine apstrakcije.
- Koristan pristup u početnom oblikovanju sustava.
- Prokazuje **nekonzistentne** i **dvosmislene** specifikacije (koje izviru iz analize zahtjeva).
- **Nema dodatnih alata** – formalna specifikacija je sama za sebe tehnika.
- U nekim slučajevima, formalna specifikacija je i jedina FM koja se koristi u industrijskim projektima

Primjer formalne specifikacije: metoda Z (čita se "zed")

- Metoda je razvijena na University of Oxford, UK (originalno Abrial 1977.)
  "Why Z?" Abrial – "Because it's the ultimate language!" ☺
- Temeljena na logici predikata prvoga reda, teoriji skupova i lambda računu.
- **Z dokument** se sastoji iz matematičkih definicija isprepletenih s engleskim tekstom.
- **Z tekst je pisan kao skup struktura – Schema**.

Značajke

- Započinje se s osnovnim definicijama i ograničenjima sustava.
- Nastavlja se s opisom akcija sustava.
- Završava se dokazivanjem različitih teorema koji garantiraju konzistentnost definicija.
- U deklarativni dio Scheme mogu se uključiti imena drugih postojećih Schema (nasljeđivanje).

Primjer formalne specifikacije: Z metoda

- **Tipovi schema:**
  - State schema – globalne izjave o sustavu.
  - Operation schema – opisuje učinak određenih operacija koje mijenjaju stanje podataka u sustavu.
  - Observation schema – opisuje dohvat informacija (podaci u sustavu se ne mijenjaju).

  - Dozvoljene su sve logičke operacije između Schema kao i kvantifikacija (egzistencijska i univerzalna).

Primjer formalne specifikacije: Z metoda

Primjer videoteke (izvor: R.Qu, Univ. Of Nottingham, UK).

- Postoji skup objekata (varijabla) *all_videos* tipa VIDEO.
- Podskup tih objekata je (varijabla) *in_stock* koji se nalaze u videoteci.
- Postoji podskup iznajmljenih videa (varijabla) *booked_out*.
- Pojedinačni video koji posjeduje videoteka mora biti u jednom od dva naprijed navedena skupa (*in_stock* ili *booked_out*).

Primjer formalne specifikacije: Z metoda

State Schema:

$[VIDEO]$ ← osnovni tip (skup) objekata

$Video_shop$ ← naziv scheme

deklaracije varijabli → $all_videos, in_stock, booked_out : \mathbb{P}\ VIDEO$

odnosi između vrijednosti varijabli → $in_stock \cup booked_out = all_videos$

svi podskupovi "powerset"

Može sadržavati i druge odnose kao npr.

$max_videos : \mathbb{N}$

$\#all_videos \leq max_videos$

Primjer formalne specifikacije: Z metoda

Operation Schema (npr.: vraćanje videa)

Video treba maknuti iz skupa *booked_out* i dodati u skup *in_stock*.

Video ne smije prije toga biti u skupu *in_stock*.

Uključi schemu ***Video_shop*** uz promjenu stanja Δ

$$
\begin{array}{|l}
\hline
\textit{Video_returned} \\
\Delta \textit{Video_shop} \\
\textit{video?} : \textit{VIDEO} \\
\hline
\textit{video?} \in \textit{booked_out} \\
\textit{booked_out}' = \textit{booked_out} - \{\textit{video?}\} \\
\textit{in_stock}' = \textit{in_stock} \cup \{\textit{video?}\} \\
\textit{all_videos}' = \textit{all_videos} \\
\hline
\end{array}
$$

preduvjet koji mora biti zadovoljen prije navedenih operacija

? = ulazne vrijednosti, ´ = nove vrijednosti

Primjer formalne specifikacije: Z metoda

Observation Schema (npr.: je li video slobodan za posudbu)

$MESSAGE :: is_in_stock \mid is_booked_out$

Uključi schemu ***Video_shop*** bez promjene stanja Ξ

$$
\begin{array}{|l}
\hline
\textit{FindVideo} \\
\Xi Video_shop \\
video? : VIDEO \\
message! : MESSAGE \\
\hline
video? \in all_videos \\
video? \in in_stock \Rightarrow message! = is_in_stock \\
video? \notin in_stock \Rightarrow message! = is_booked_out \\
\hline
\end{array}
$$

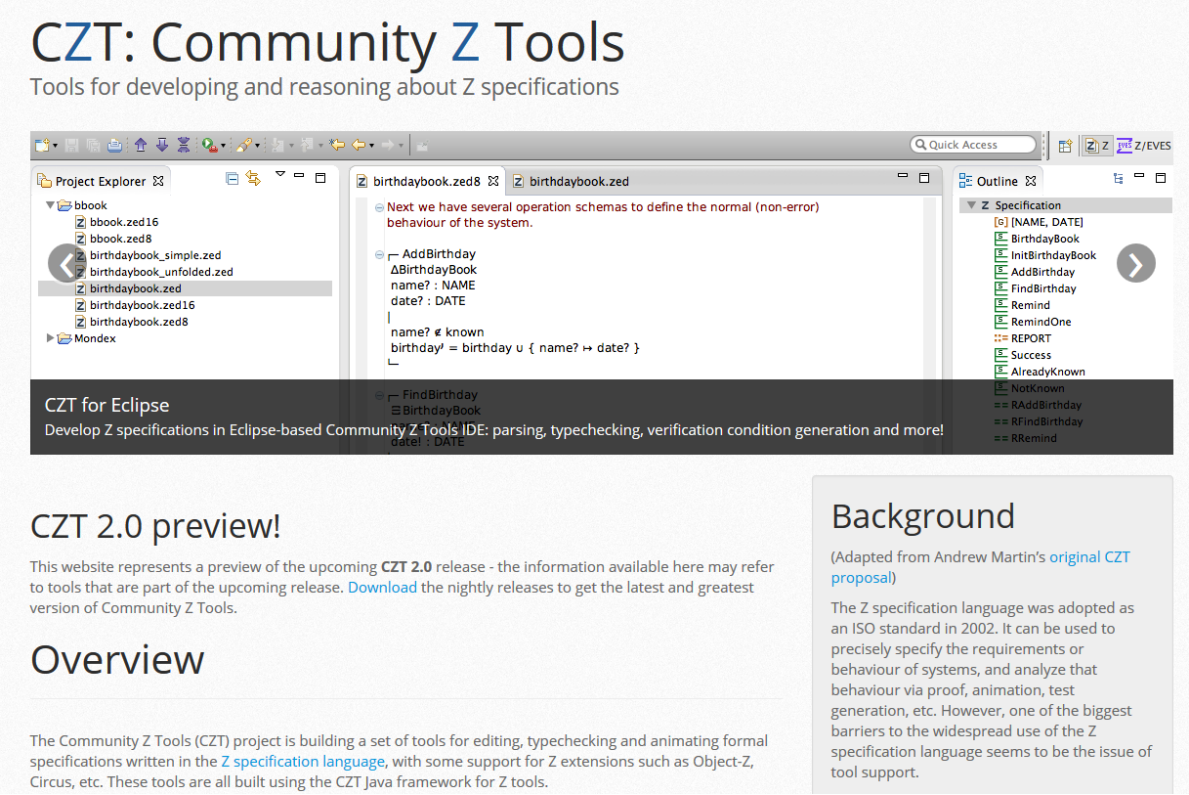
preduvjeti i operacije

Primjer formalne specifikacije: Z metoda

http://czt.sourceforge.net/

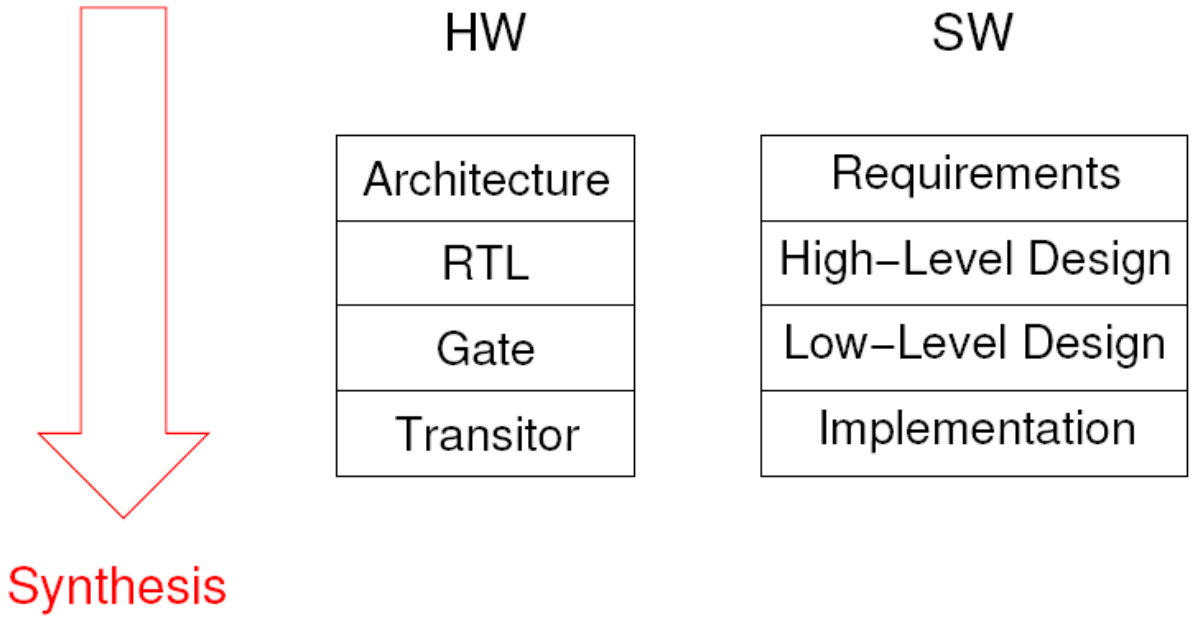
https://sourceforge.net/projects/zwordtools/

https://abz2023.loria.fr/



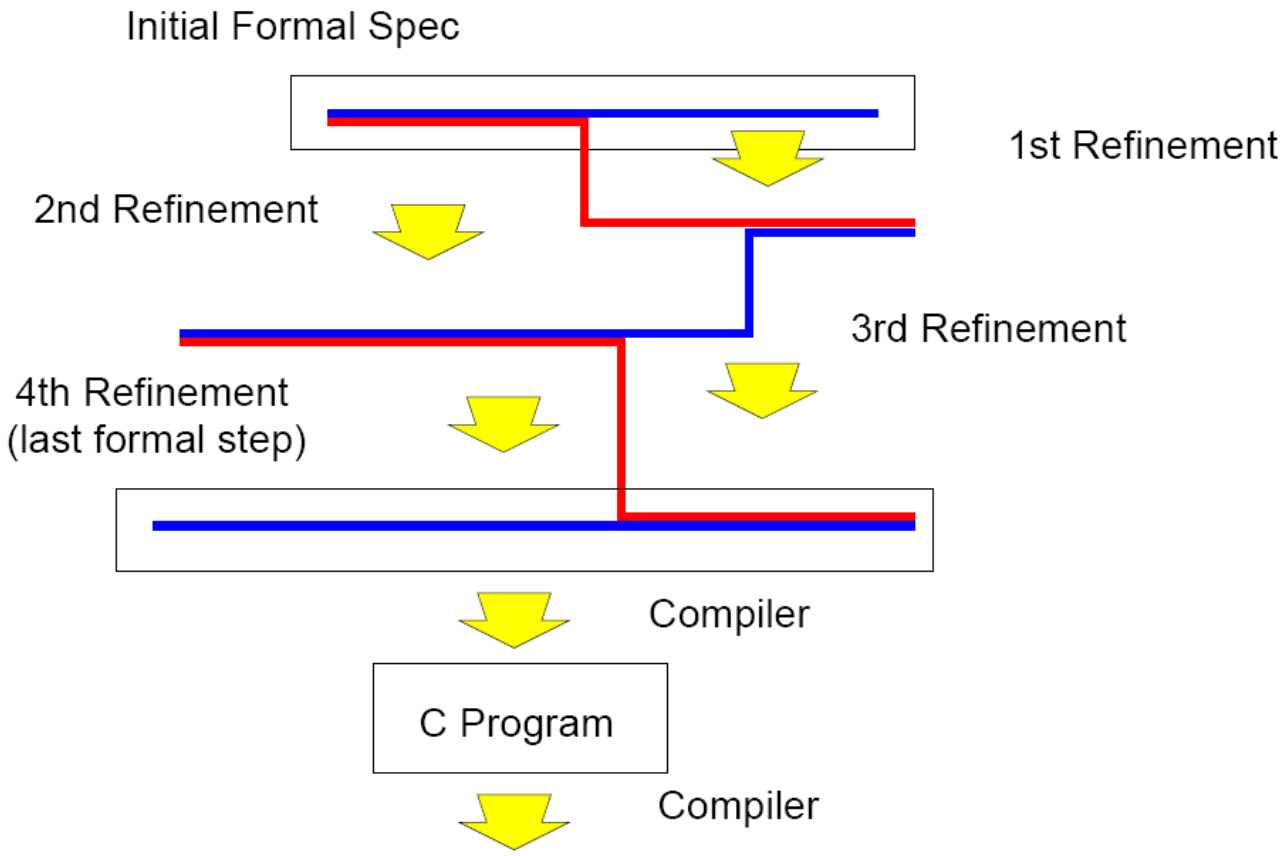
Formalna sinteza

Oblikovanje sustava odozgo prema dolje – od najveće razine apstrakcije do implementacije (rafiniranje), korištenjem formalizama za verifikaciju, pri čemu se raspodjeljuje verifikacijski posao na male zadatke



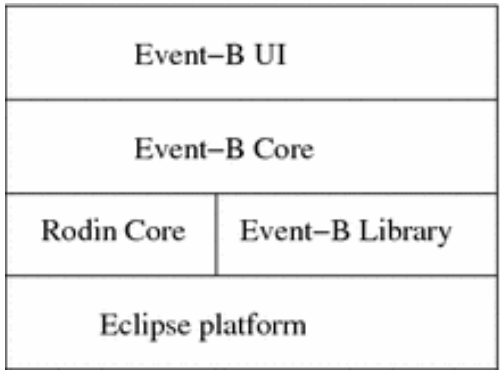
Formalna sinteza

- Ideja:



Primjer formalne sinteze – B-metoda

- **B-metoda i Event-B** (+ platforma **Rodin**)
  - http://www.event-b.org/index.html
- Razvijaju se od 1988. do danas
- Event-B je novija od B-metode i danas se više koristi zbog manje složenosti notacije



- **Event-B** koristi:
  - **Teoriju skupova** kao notaciju za modeliranje
  - **Rafiniranje** za predstavljanje sustava na različitim razinama apstrakcije
  - **Matematičke dokaze** za provjeru konzistentnosti između različitih razina kao način deduktivnog rasuđivanja
- Široko korištena notacija i platforma u industrijskim i istraživačkim projektima: *http://wiki.event-b.org/index.php/Industrial_Projects*

Primjer formalne sinteze – B-metoda

- Svaki **model** u Event-B se sastoji od statičkog dijela – **konteksta** i dinamičkog dijela – **stroja**
- Kontekst sadrži:
  - **konstante** i
  - **aksiome** koji opisuju svojstva konstanti
- Stroj se sastoji od:
  - **stanja** – opisana **varijablama**
  - **događaja** koji specificiraju evoluciju stanja
- Notacija uključuje više ključnih riječi (npr. *when*, *then*, *end*...) koje su samo delimiteri da bi tekstualna reprezentacija modela bila čitljiva
- Više o notaciji:

  http://www3.hhu.de/stups/handbook/rodin/current/html/mathematical_notation_introduction.html

Primjer formalne sinteze – B-metoda

- **Varijable** su jednostavni matematički objekti koji mogu biti
  - Skupovi
  - Binarne relacije
  - Funkcije
  - Brojevi
- Varijable mogu biti ograničene **invarijantama** – logičkim tvrdnjama koje moraju vrijediti pri svakoj promjeni vrijednosti varijable

Primjer formalne sinteze – B-metoda

- **Događaji** se sastoje od **preduvjeta** (engl. *guard*) i od **akcije** (engl. *action*), a mogu imati i **parametre**
- Preduvjet nužno treba biti ispunjen da bi se događaj pokrenuo
- Akcija određuje način na koji varijable stanja evoluiraju pri pojavi događaja
  - Definira skup pridruživanja koja modificiraju stanje
- Samo jedan događaj može biti izabran za izvođenje u nekom trenutku – izbor je nedeterministički

Primjer formalne sinteze – B-metoda

Primjer izrade sustava za provjeru registriranih korisnika pri ulasku i izlasku iz zgrade

1. Stvorimo novi kontekst i dodamo skup objekata *USER* u taj kontekst
2. Stvorimo novi stroj i dodamo varijablu *register* u taj stroj, koja predstavlja skup registriranih korisnika
3. Dodamo invarijantu da bi *register* definirali kao skup korisnika:

   inv1 $register \subseteq USER$

   Automatski, tip varijable *register* je **P**(*USER*) – partitivni skup od *USER*
4. Dodamo događaj `Register` za dodavanje novog korisnika u *register*:

   `Register` ≜ **any** *u* **where**

   grd1 $u \in USER \backslash register$ // preduvjet: *u* je u *USER* ali ne u *register*

   then

   act1 $register := register \cup \{u\}$ // akcija: *register* je pojačan s novim članom

   end

Primjer formalne sinteze – B-metoda

5. Dodamo događaj inicijalizacije `Initialization`: *register* := ∅
6. Dodamo varijable ljudi koji su u zgradi (*in*) i izvan zgrade (*out*) i dodamo ograničenja:

   inv2 *in* ⊆ *register* // primijetiti da je *register* varijabla

   inv3 *out* ⊆ *register*
7. Dodamo razumljivu invarijantu da korisnik ne može biti istovremeno u zgradi i izvan nje:

   inv4 *in* ∩ *out* = ∅

Ovakav model sada iziskuje šest **matematičkih dokaza** (engl. *proof obligations*), tri se tiču toga da inicijalizacija ustanovljava inv2 – inv4, a tri se tiču toga da događaj `Register` održava invarijante inv 2 – inv4; svi dokazi prolaze

Primjer formalne sinteze – B-metoda

8. Sada trebamo dodati događaje za ulazak u zgradu i izlazak iz nje. Ako pokušamo ovako za ulazni događaj:

`Enter` ≙ **any** *u* **where**

grd1 $u \in out$

then

act1 $in := in \cup \{u\}$

end

Onda dokaz pada na invarijanti inv4 $in \cap out = \emptyset$

Hyp1 : $in \cap out = \emptyset$

Hyp2 : $u \in out$

$\vdash$

Goal : $(in \cup \{u\}) \cap out = \emptyset$

Greška je ili u postavljenoj invarijanti ili u događaju `Enter`. Očito, u ovom slučaju je greška u događaju `Enter`, tako da:

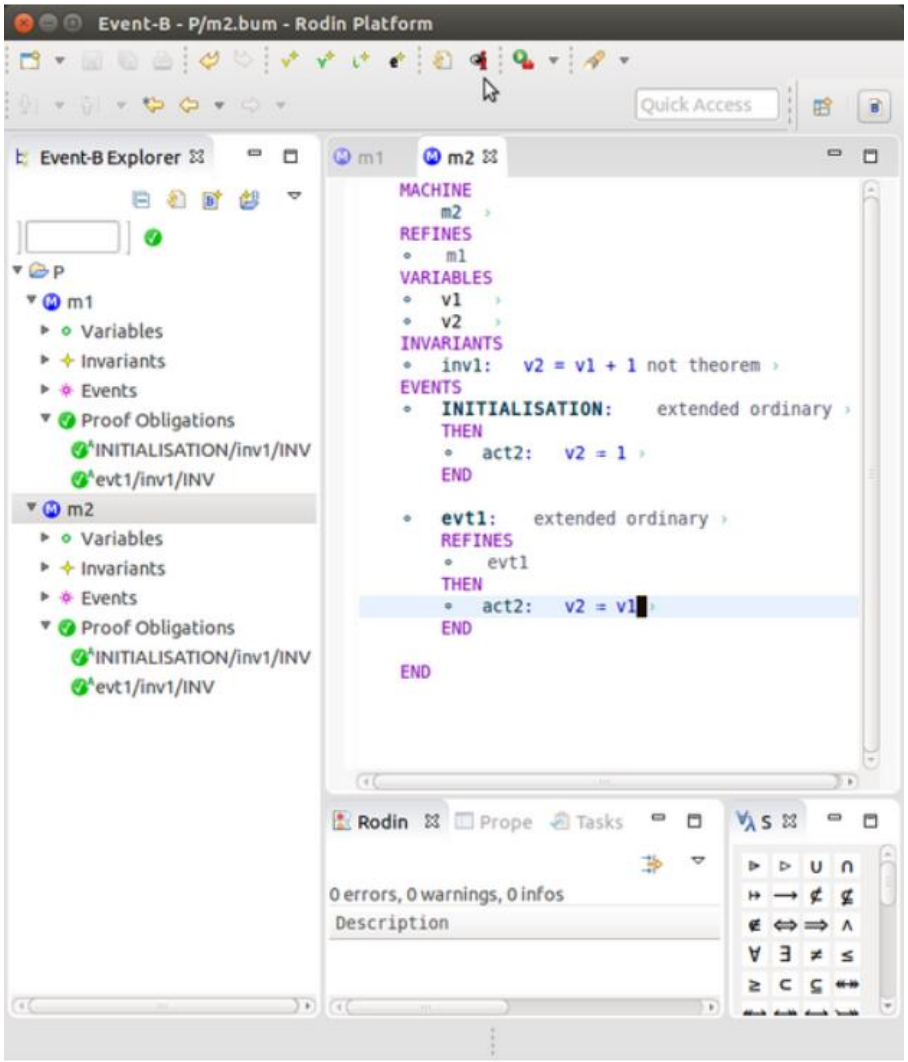
9. Dodajemo novu akciju događaju `Enter`:

act2 $out := out \setminus \{u\}$

i rafiniranje modela ide dalje...

Primjer formalne sinteze – B-metoda

- Platforma Rodin izgrađena je na Eclipse-IDE



Primjer formalne sinteze – Prevoditelj (engl. *compiler*), Izvor: A. Artale, Univ. of Bolzano, Italy

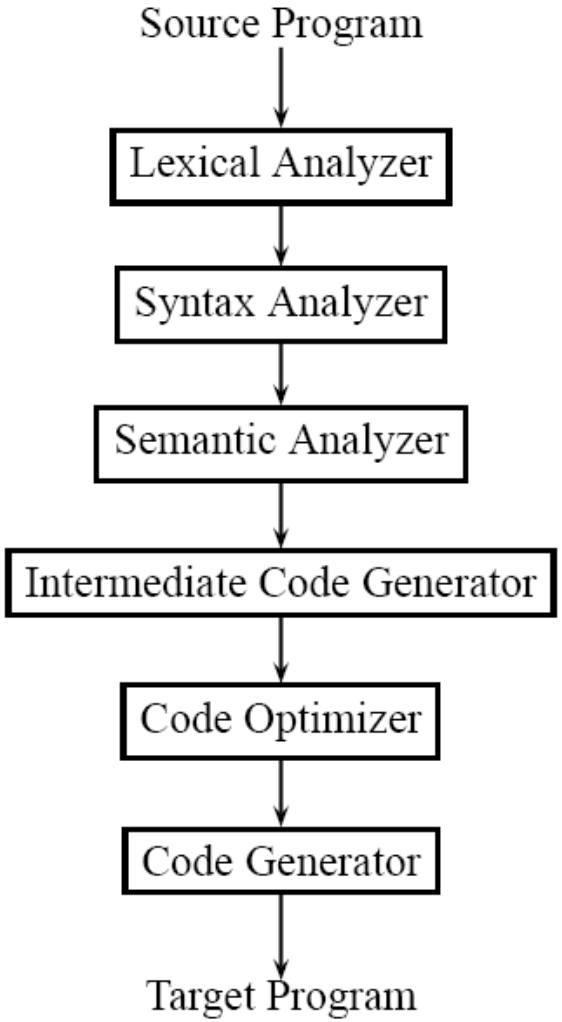
- Prevoditelji preslikavaju programe iz jednog jezika (izvorni jezik) u drugi jezik (ciljni jezik)
- Dodatno, prevoditelju su nužni drugi programi za generiranje strojnog, izvršnog koda
- Interpreteri tumače i izvode program bez njegovog prethodnog prevođenja u strojni kod

Slijedni proces prevođenja:

Izvorni program (*engl. source code*) -> Prevoditelj (engl. *Compiler*) -> Ciljni asemblerski program -> Asembler -> Relokatibilni strojni kod -> Povezivanje (engl. *Linker*) -> Izvršni kod -> Punjenje (engl. *Loader*) -> Završno razrješenje adresa i program pokrenut u memoriji

Primjer formalne sinteze – Prevoditelj (engl. *compiler*)

- Arhitektura prevoditelja
  - Analiza (dekompozicija izvornog programa u dijelove i kreiranje prijelazne reprezentacije).
  - Sinteza (generiranje ciljnog programa temeljem prijelazne reprezentacije).
- Analiza ima sljedeće faze:
  - Leksička analiza
  - Sintaksna analiza
  - Semantička analiza
- Sinteza ima sljedeće faze:
  - Generiranje prijelaznog koda
  - Optimizacija koda
  - Generiranje ciljnog koda



Primjer formalne sinteze – Prevoditelj (engl. *compiler*)

- Leksička analiza i izgradnja tablice simbola
  - Izdvajanje slijeda znakova (leksema) i grupiranje u leksičke značke (engl. *tokens*) određenog tipa.

Npr:

```
position = initial + rate * 60
```

ID - Identifier

Lexeme	Token
position	ID
=	=
initial	ID
+	+
rate	ID
*	*
60	NUM

Primjer formalne sinteze – Prevoditelj (engl. *compiler*)

- Sintaksna analiza = parsiranje (grupiranje znački u gramatičke fraze predstavljene stablom parsiranja).
- Sintaksna analiza daje hijerarhijsku strukturu izvornog programa izraženu preko rekurzivnih pravila ili produkcija (engl. *productions*).
- Semantička analiza provjerava konzistentnost tipa operanda.
- Primjer sintaksnih pravila za naredbe pridruživanja:

$$
\begin{aligned}
< assignment > &\rightarrow \text{ID} \text{ “} = \text{”} < expr > \\
< expr > &\rightarrow \text{ID} \mid \text{NUM} \mid < expr >< op >< expr > \mid (< expr >) \\
< op > &\rightarrow + \mid - \mid * \mid /
\end{aligned}
$$

Primjer formalne sinteze – Prevoditelj (engl. *compiler*)

- Generiranje prijelaznog koda (međukoda).
  - Najčešće slijed naredbi s najviše tri operanda takav da:
    - Postoji najviše **jedan operator** pored pridruživanja
    - Generiraju se privremena imena za izračunavanje prijelaznih operacija

- Primjer: `position = initial + rate * 60`

  `id1` `id2` `id3`

```
temp1 = inttoreal(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

$\xrightarrow{\text{optimizacija}}$

```
temp1 = id3 * 60.0
id1 = id2 + temp1
```

Primjer formalne sinteze – Prevoditelj (engl. *compiler*)

- Generiranje (asemblerskog) koda
  - Odabir memorijskih lokacija za varijable (id1, id2, id3).
  - Naredbe višeg jezika preslikavaju se u sekvence asemblerskih naredbi.
  - Varijable i međurezultati pridjeljuju se memorijskim lokacijama.
- Primjer: `position = initial + rate * 60`

  id1 id2 id3

```
MOVF id3, R2        //F=floating_point,R=register
MULF #60.0, R2      // result in R2
MOVF id2, R1
ADDF R2, R1         // result in R1
MOVF R1, id1
```

Primjer formalne sinteze – Prevoditelj (engl. *compiler*)

Prevođenje u izvršni kod (asembler, povezivanje, punjenje).

- Asemblerski kod je mnemonička, čovjeku čitljiva inačica izvornog programa gdje se koriste simboli umjesto memorijskih lokacija (iz tablice simbola) i operacija (kodna tablica). Asemblerski kod se zatim pomoću programa asemblera pretvara u objektni strojni kod koji više nije čovjeku čitljiv.

- U fazi povezivanja (engl. *link*) spaja se skup neovisno oblikovanih modula u obliku objektnog koda i datoteka knjižnica (engl. *library files*) u jedinstveni izvršni program.

- Generirani strojni kod je **relokatibilan** te se u fazi punjenja (engl. *load*) određuju apsolutne memorijske adrese (osim statičkih apsolutnih adresa ulazno-izlaznih naprava i sličnih entiteta) i program se pokreće u memoriji.

Ostale značajne metode formalne specifikacije i sinteze

- **TLA i TLA$^+$ (Temporal Logic of Actions)** (+ alat TLA Toolbox) – **L. Lamport (Microsoft)**
  - https://lamport.azurewebsites.net/tla/tla.html
- **Alloy – D. Jackson (MIT)**
  - http://alloytools.org/
- **ASM (Abstract State Machines) – Y. Gurevich (Microsoft)**
  - http://web.eecs.umich.edu/gasm/
  - http://research.microsoft.com/en-us/projects/asml/
- **VDM i VDM++** (*Vienna Development Method*) (+ alat Overture)
  - http://overturetool.org/
- **SDL - Specification and Description Language**
  - http://www.sdl-forum.org/

Formalna verifikacija - FV

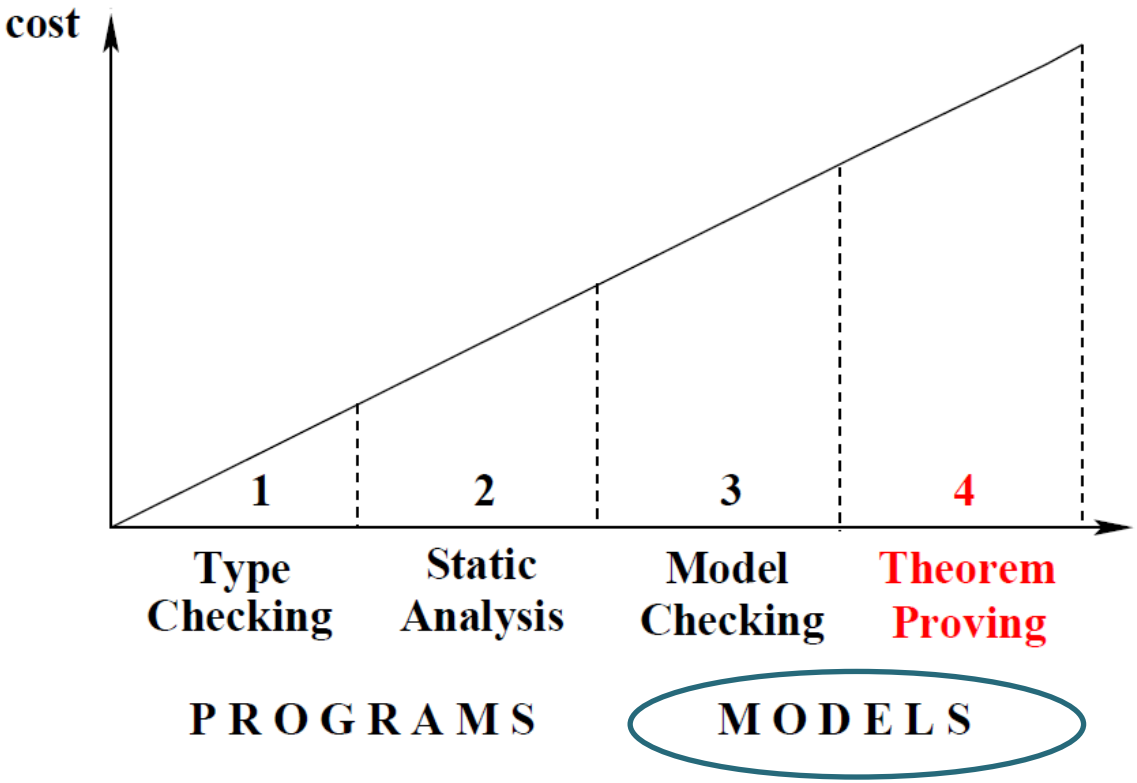
- **Definicija:** FV je proces analize sustava kojim dokazujemo da **implementacija** sustava posjeduje svojstva koju zahtijeva formalna **specifikacija** sustava

- *"Program **testing** can be used to show the presence of bugs, but never to show their absence."* [Dijkstra, 1972]

- Nadamo se da "bug" ne postoji (nije nam poznato), ali želimo biti sigurni. Provodi se dokazivanje ili opovrgavanje ispravnosti algoritama u odnosu na zadanu formalnu specifikaciju ili svojstvo.
- Prolazi se svim putevima kroz program, a ne samo nekima kao kod ispitivanja.

Formalna verifikacija - FV

- FV provodi se u najvećem broju slučajeva nad **modelima** stvarnih programa (modelima implementacije)



Formalna verifikacija - FV

Metode formalne verifikacije

Provjera modela (engl. *Model checking*):

Provjerava zadovoljava li model implementacije zadano obilježje (specifikaciju).

Dokazivanje teorema (engl. *Theorem proving*):

Dokazuje u nekom logičkom formalizmu je li neka specifikacija logička posljedica implementacije (skupa formula).

Provjera ekvivalentnosti (engl. *Equivalence checking*):

Uspoređuje novo oblikovani model ili implementaciju s izvornim modelom ili implementacijom da bi se ustanovila logička ekvivalentnost.

Provjera tvrdnje (engl. *Assertion-based verification*):

Provjerava se da neko specifično svojstvo ili uvjet mora uvijek biti zadovoljeno.

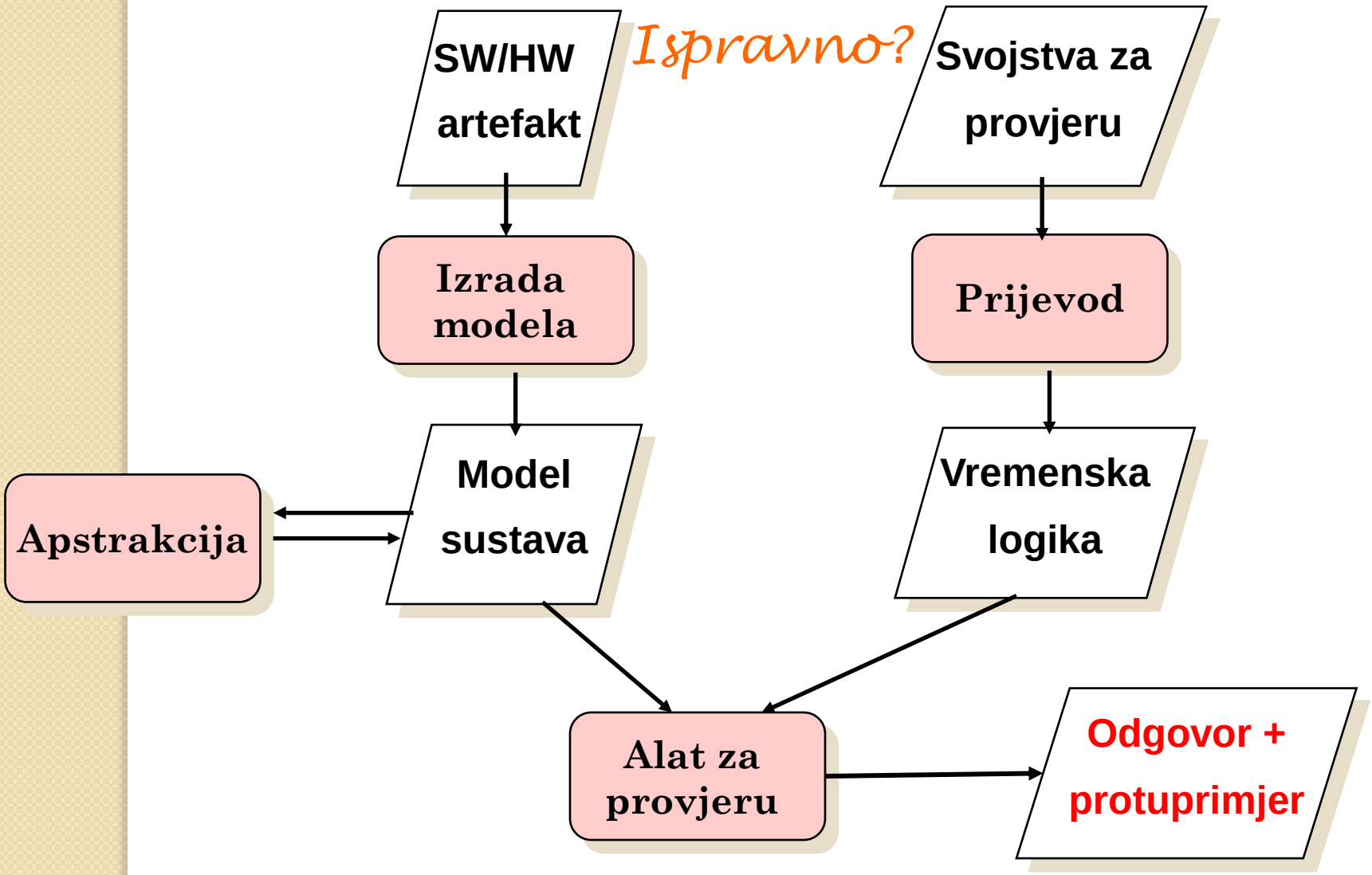
Provjera modela

- **Provjera modela** – automatizirana metoda provjere reaktivnih sustava modeliranih strojevima s konačnim brojem stanja na zadanu specifikaciju (obilježje, ponašanje).
- Provjerava se zadovoljava li u logičkom smislu (model) implementacije zadano obilježje (specifikaciju)
- Temelji se na potpunom pretraživanju prostora stanja (niti jedno stanje ne ostaje neispitano – razlika prema ispitivanju).
- Moguć vrlo veliki prostor stanja (> 10^{150}).

Provjera modela

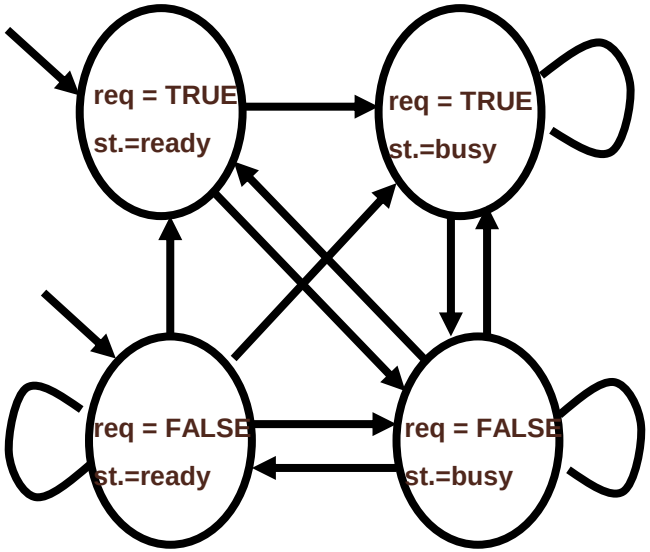
- **Proces provjere modela:**
  - Implementaciju prikazati modelom u formalizmu svojstvenom alatu za provjeru
  - Specifikaciju ponašanja prikazati nekom **logikom**, najčešće vremenskom logikom
  - Verifikacija je **automatizirana** danim alatom.
- Varijante:
  - **Ograničena provjera modela** (engl. *bounded model checking*) – provjera (modela) implementacije do nekog *k*-tog stanja
  - Provjera modela za sustave za rad u stvarnom vremenu – najčešće vremenski automati, vidjeti npr. Uppaal: http://www.uppaal.org/

FV: Provjera modela



FV: Provjera modela

Primjer modela implementacije opisanog u alatu za FV (NuSMV) – vrijedi samo za taj konkretni alat




```
MODULE main
VAR
   request:    boolean
   status:    {ready, busy}
ASSIGN
   init(status)  := ready;
   next (status)  := case
                     request: busy;
                     TRUE: {ready, busy}
                     esac;
```

Jedna formalna specifikacija u vremenskoj logici:

AG(request ⇒ AF (status = busy))

FV: Provjera modela

2007 Turing Award Winners Announced for their groundbreaking work on Model Checking

Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis are the recipients of the 2007 A.M. Turing Award for their work on an automated method for finding design errors in computer hardware and software. The method, called **Model Checking**, is the most widely used technique for detecting and diagnosing errors in complex **hardware and software** design. It has helped to improve the reliability of complex computer chips, systems and networks.

FV: Dokazivanje teorema

- Formalna verifikacija programa – provjera da program doista radi ono što bi trebao
- **Dokazivanje** rada programa – obrazloženje o programu kako bi se zajamčila točnost
- Dokazivanje rada programa ima smisla samo ako imamo neku vrstu **specifikacije** onoga što bi program trebao raditi
- Mnoge su specifikacije napisane na prirodnom jeziku što može dovesti do nepreciznosti i nesporazuma

- Za stvarne programe često se koristi **oblikovanje prema ugovoru** (engl. *design by contract*)
- Specifikacije su zadane kao **preduvjeti** i **postuvjeti** te **algebarska specifikacija invarijanti**

FV: Dokazivanje teorema

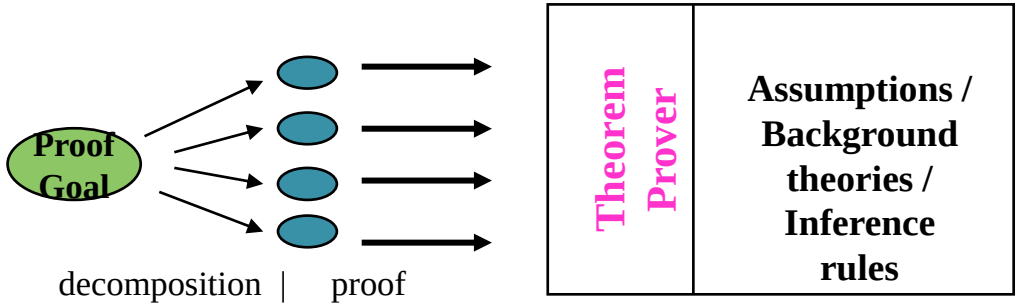
Zadatak dokazivanja teorema:

Dokazati da je neka **specifikacija logička posljedica implementacije** u nekom odabranom logičkom formalizmu.

Ulazi u sustav za dokazivanje teorema:

- Logičke formule koje opisuju implementaciju (tzv. aksiomi).
- Logička formula koja opisuje specifikaciju
- Pretpostavke o domeni problema (Npr. Vcc=True).
- Teorija (pravila dokazivanja, ranije dokazani teoremi, …).

Izlaz sustava: Dokaz da je **impl $\models$ spec**



FV: Dokazivanje teorema

- Metodu nije jednostavno primijeniti u industriji jer većina inženjera nema znanja o formalnoj matematičkoj logici.
- Automatizacija je ograničena, postupak često traži ručno vođenje.
- Potrebno duboko poznavanje alata za verifikaciju.

Primjer:

Predikatna logika u verifikaciji programa dokazivanjem teorema:

Stanje programa: PSTATE(tekuća_naredba, lista_varijabli)

Svaka naredba se mora preslikati u logiku. Npr. ASSIGN:

ASSIGN(xinstr, zassignto, wexpr, yinstr)

xinstr – tekuća naredba, zassignto – varijabla u koju se upisuje,

wexpr – izraz evaluacije prije zapisivanja, yinstr – sljedeća naredba

Pravila, npr. "Program u stanju 1 naredbom ASSIGN prelazi u stanje 2":

$(PSTATE_1 \wedge ASSIGN) \Rightarrow PSTATE_2$

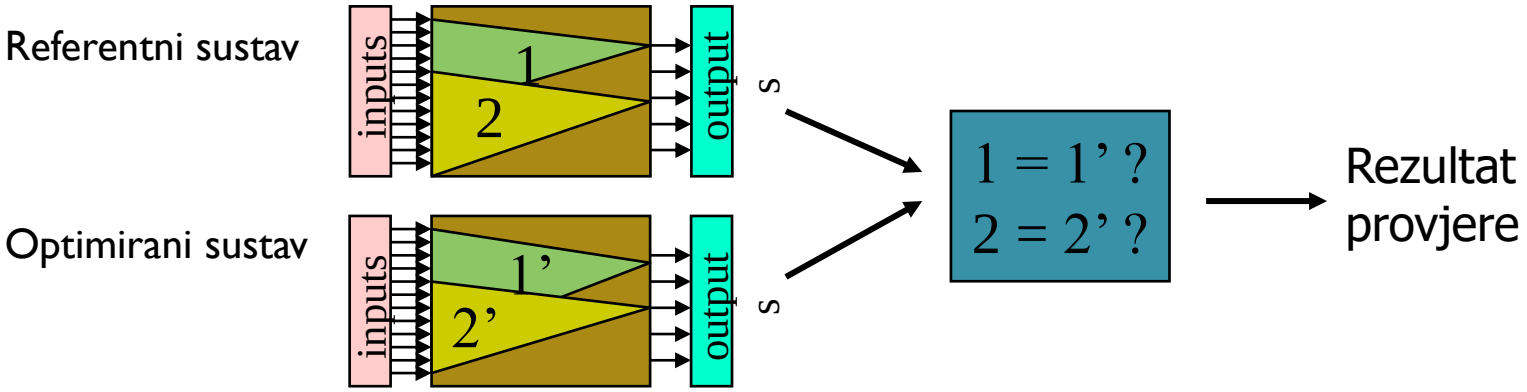
FV: Dokazivanje teorema

- Postoje mnogi razvijeni jezici i alati za dokazivanje teorema, primjerice:
  - **KeY-Hoare** – dokazivanje programa specifikacijama u Hoareovoj logici
    - http://i12www.ira.uka.de/key/download/
  - **Dafny** – programski jezik za pisanje sigurnih programa
    - https://marketplace.visualstudio.com/items?itemName=dafny-lang.ide-vscode
    - https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml220.pdf
  - **Verifast** – dokazivanje C i Java programa – separacijska logika
    - https://people.cs.kuleuven.be/~bart.jacobs/verifast/
  - **Coq** – dokazivanje različitih matematičkih svojstava – metoda računanja induktivnih konstrukcija = logika višeg reda + funkcijski jezik
    - https://coq.inria.fr/
  - **Spark2014** – formalna verifikacija programa pisanih u jeziku Ada 2012
    - https://www.adacore.com/about-spark
  - **Isabelle/HOL** – iskazivanje matematičkih formula u formalizmu i dokazivanje formula – logika višeg reda
    - https://www.cl.cam.ac.uk/research/hvg/Isabelle/index.html
  - …

FV: Provjera ekvivalentnosti

Kombinacijska provjera

Provjerava funkcijsku ekvivalentnost sustava koji nemaju memoriju (stanja).



Sustav se razbije na "logičke konuse" i uspoređuju se izlazi za jednake ulaze.

Značajno za sklopovlje pa nećemo obrađivati detaljnije u ovom predmetu.

FV: Provjera tvrdnje

- Opisuje se očekivani ili neočekivani uvjet (tvrdnja – "assertion") koji mora biti zadovoljen u implementaciji
- Tvrdnju se često uključuje u sustav koji ispitujemo
- Tvrdnja se najčešće opisuje proširenjima u jeziku implementacije, najčešće naredbom `assert(neki_uvjet)` koja mora biti ispunjena ili baca pogrešku
- Provjera tvrdnje se obično treba posebno uključiti prilikom pokretanja

```
// Java program to demonstrate syntax of assertion

class Test
{
    public static void main( String args[] )
    {
        int value = 15;
        assert value >= 20 : "Underweight";
        System.out.println("value is "+value);
    }
}
```

Zaključci o formalnim metodama (FM)

- Pojam ispravnosti (engl. *correctness*) je temeljni intelektualni izazov u industriji složenih sklopovskih i programskih proizvoda.
- Formalna verifikacija je rigorozna (matematički utemeljena) demonstracija ispravnosti.
- Dok je ručna verifikacija dostatna za male kritične sustave, složeni sustavi traže automatizirane postupke FV.
- Automatizirani postupci FV počivaju na skupu efikasnih alata (laganih za primjenu i brzih u donošenju odluke).
- Alati za FV potiču još brži razvitak gospodarstva temeljenog na složenim sklopovskim i programskim proizvodima.

FVPP: Teorijske osnove formalne verifikacije provjerom modela

Propozicijska logika

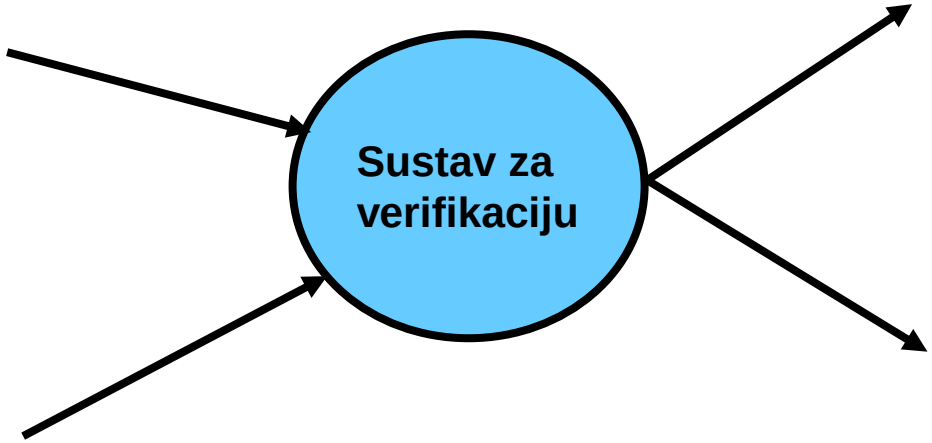
Predikatna logika

Pripremio: izv. prof. dr. sc. Alan Jović

Ak. god. 2022./2023.

Provjera modela (engl. *model checking*)

I **= Implementacija (model sustava koji se verificira). Izraženo povezanim strojevima s konačnim brojem stanja (FSM).**



DA = model sustava logički zadovoljava specifikaciju

NE (+ ispis traga (engl. *trace*) pogrešnog izvođenja programa)

S **= Specifikacija (željeno ponašanje). Izraženo najčešće u vremenskoj logici.**

Simbolički opisujemo:

$$\boxed{I \models S}$$

Logika u provjeri modela

- Provjerom modela nastoji se dokazati **logička zadovoljivost** ("model implementacije zadovoljava specifikaciju") – za to su potrebna znanja iz sljedećih područja:
  - **Formalna (matematička) logika**
  - **Modeliranje implementacije** strojevima s konačnim brojem stanja (u kontekstu formalne verifikacije model je tzv. Kripkeova struktura)
  - **Izražavanje specifikacije** (željenog ponašanja) vremenskom logikom kao proširenjem klasične matematičke logike

Formalna (matematička) logika

- Formalne logike su formalni jezici koji predstavljaju informaciju na način da se mogu automatizirano izvoditi "zaključci".
  - **Sintaksa** definira strukturu rečenice u jeziku.
  - **Semantika** definira značenje rečenica (definira istinitost rečenice u svijetu u kojem ju promatramo).

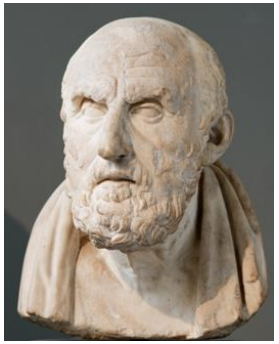
- Postoji mnogo logika:
  - **Propozicijska i predikatna logika**
  - Logike višega reda
  - Modalne logike
    - Epistemička logika
    - **Vremenska logika**
    - …
  - Opisna logika
  - Nemonotona logika
  - …

u ovom kolegiju

Propozicijska logika (logika izjava, sudova, iskaza, tvrdnji)

Engl. *propositional logic, propositional calculus*

Hrizip iz Solija



Propozicijska logika - sintaksa

- Propozicijska logika preslikava deklarativne rečenice (koje mogu biti istinite ili lažne) u sustav simbola. Naprimjer: "Sokrat je mudar." preslikava se u simbol P.
- Sustav propozicijske logike sastoji se od:
- PS: P, Q, ... PS je prebrojiv skup atoma, simboličkih varijabli, simbola
- Logički operatori (vezice):

$\neg$	**(ne, not, ~)**	**negacija**
$\wedge$	**(i, and, &)**	**konjunkcija**
$\vee$	**(ili, or, |)**	**disjunkcija**
$\Rightarrow$	**(ako, if, $\supset$, $\rightarrow$)**	**implikacija**
$\Leftrightarrow$	**(akko, iff, $\equiv$, $\leftrightarrow$)**	**ekvivalencija**

- Rezervirani simboli:

F	**(false, $\varnothing$, 0, $\perp$)**	**konstanta (neistinitost)**
T	**(true, 1)**	**konstanta (istinitost)**
(), .		**znakovi zagrada, zareza i točke**

- **Definicija** (rekurzivno) ispravno formiran (wff) složeni iskaz, ili formula:

1. Svaki atom je formula.
2. Ako su P i Q formule, onda su formule: $(\neg P)$, $(\neg Q)$, $(P \wedge Q)$, $(P \vee Q)$, $(P \Rightarrow Q)$, $(P \Leftrightarrow Q)$.

Propozicijska logika - semantika

- Pridruživanje obilježja istinitosti (T, F) atomičkim simbolima = **Interpretacija (I)**
- **I: PS → BOOL**

  gdje je BOOL = { T, F }, tj. funkcija s kodomenom T ili F (istinito ili lažno).

Semantika dvaju složenih atomičkih simbola

- prikazuje se istinitosnom tablicom.
- 2 simbola = 2^2 = 4 interpretacije, 2^4 = 16 istinitosnih tablica = funkcija

- Neke važnije tablice istinitosti za povezivanje dvaju simbola:

	P	Q	implikacija (P ⇒ Q)	ekvivalencija (P ⇔ Q)	kontradikcija (), ⊥	tautologija ⊤
I_1 :	T	T	T	T	F	T
I_2 :	T	F	F	F	F	T
I_3 :	F	T	T	F	F	T
I_4 :	F	F	T	T	F	T

Semantička pravila – izračunavanje istinitosti formule

- Neka su: P_1, P_2 istinite, Q_1, Q_2 neistinite, = interpretacija
  A bilo koja formula (istinita ili ne).

  Istinite su formule:

  $\neg Q_1$
  $(P_1 \wedge P_2)$
  $(P_1 \vee A)$
  $(A \vee P_1)$
  $(A \Rightarrow P_1)$
  $(Q_1 \Rightarrow A)$
  $(P_1 \Leftrightarrow P_2)$
  $(Q_1 \Leftrightarrow Q_2)$
  $(A \vee \neg A)$ **- tautologija**

  Neistinite su formule:

  $\neg P_1$
  $(Q_1 \wedge A)$
  $(A \wedge Q_1)$
  $(Q_1 \vee Q_2)$
  $(P_1 \Rightarrow Q_1)$
  $(P_1 \Leftrightarrow Q_1)$
  $(Q_1 \Leftrightarrow P_1)$
  $(A \wedge \neg A)$ **- kontradikcija**
  () **- prazna formula**

- Primjer izračunavanja istinitosti složene formule s 3 propozicijska simbola P, Q, R:
  $(Q \vee (((\neg Q) \wedge P) \Rightarrow R))$
- **Interpretacija** (3 simbola povlači 2^3 mogućih interpretacija)
- Neka je jedna interpretacija I: P=T, Q=F, R=F,
- Izračunavanje (**evaluacija**) temeljem osnovnih tablica istinitosti i računajući "iznutra prema van" daje ovoj formuli neistinitu vrijednost.

- **Semantika uključuje interpretaciju i evaluaciju.**

Pravila ekvivalencije

- Definicija: Dvije formule su semantički **ekvivalentne** ili **jednake** ako imaju jednaku (istu) istinitosnu vrijednost **za svaku interpretaciju**.
- Provjera koincidencije istinitosnih tablica **nije u općem slučaju dovoljna** jer formule ne moraju sadržavati iste simbole. Vidjeti za opći slučaj sl. 20.

$(A \wedge \neg A) = ()$	**kontradikcija**
$(\neg(\neg A)) = A$	**dvostruka negacija**
$(A \wedge A) = A$	**jednaka važnost (idempotencija)**
$(A \vee A) = A$	**jednaka važnost**
$(A \vee B) = (B \vee A)$	**komutativnost**
$(A \wedge B) = (B \wedge A)$	**komutativnost**
$((A \vee B) \vee C) = (A \vee (B \vee C))$	**asocijativnost**
$((A \wedge B) \wedge C) = (A \wedge (B \wedge C))$	**asocijativnost**
$(A \wedge (B \vee C)) = ((A \wedge B) \vee (A \wedge C))$	**distributivnost**
$(A \vee (B \wedge C)) = ((A \vee B) \wedge (A \vee C))$	**distributivnost**
$(\neg(A \vee B)) = ((\neg A) \wedge (\neg B))$	**De Morganov zakon**
$(\neg(A \wedge B)) = ((\neg A) \vee (\neg B))$	**De Morganov zakon**
$(A \Rightarrow B) = ((\neg A) \vee B)$	**eliminacija uvjeta**
$(A \Leftrightarrow B) = ((A \Rightarrow B) \wedge (B \Rightarrow A))$	**eliminacija dvostrukog uvjeta**
$(A \Rightarrow B) = ((\neg B) \Rightarrow (\neg A))$	**transpozicija**

Formalan sustav

- Matematička ili formalna logika daje sustav zaključivanja u kojem je "logički izveden" zaključak barem tako dobar kao polazne pretpostavke.
- Temelj formalne logike: **formalan sustav**

- Definiramo formalan sustav kao dvojku: $\{\Gamma, L\}$ u odabranoj logici gdje je
  - Γ – skup ispravno definiranih (formiranih) formula (wff)
  - **L** – konačan skup pravila zaključivanja

Temeljna pravila zaključivanja L

- Generiraju dodatne istinite formule (mehanički) iz početnih formula – **aksioma** formalnog sustava bez razumijevanja konteksta (značenja).
- Pogodna za strojnu primjenu.

- Ako P=**T**, Q=**T**, generiraj **(P∧Q) = T** (uvođenje ∧ = ∧i)
- Ako P=**T**, (P⇒Q)=**T**, generiraj **Q = T** ("modus ponens")
- Ako ¬Q=**T**, (P⇒Q)=**T**, generiraj **¬P** ("modus tolens")
- Ako (P∧Q)=**T**, generiraj **(Q∧P) = T** (komutativnost ∧)
- Ako (P∧Q)=**T**, generiraj **P = T, Q = T** (eliminacija ∧ = ∧e)
- Ako P=**T** (odnosno Q=**T**), generiraj **(P∨Q)** (uvođenje ∨ = ∨i)
- Ako [¬(¬P)]=**T**, generiraj **P = T** (eliminacija ¬ = ¬e)

- Kroz primjere za vježbu pokazat će se još i pravilo "eliminacija ∨" kao i pravilo "uvođenje ⇒" , vidjeti i zadnji slajd

Terminologija formalnih sustava

Dedukcija (engl. deduction)

- Sekvencija formula $\{\omega_1, \omega_2, \ldots, \omega_n\}$ ili pojedina formula ω_i je **dedukcija (dokaz)** iz skupa formula Γ ako se već nalazi u skupu formula Γ (tada se naziva aksiom) ili se može izvesti iz Γ korištenjem pravila zaključivanja **L**.

$\Gamma \vdash_L \{\omega_1, \omega_2, \ldots, \omega_n\}$	sekvencija formula je dedukcija od Γ
$\Gamma \vdash_L \omega_i$	formula ω_i je dedukcija od Γ

Primjeri

- Neka skup Γ sadrži dvije istinite formule: $\Gamma = \{ P, (P \Rightarrow Q) \}$

  Korištenjem pravila **"Modus ponens"** (iz skupa dopustivih pravila **L**), izvodimo da je istinita nova formula Q, te je ta formula Q dedukcija (dokaz) skupa Γ.
- Neka skup Γ sadrži tri istinite formule: $\Gamma = \{P, Q, (Q \wedge R)\}$

  Formule P, Q, i $(Q \wedge R)$ su aksiomi, a ujedno i dedukcije, dok je formula R dedukcija, jer to daje pravilo eliminacije $\wedge$.

Terminologija formalnih sustava

Teorem (engl. *theorem*)

- Formula ω_i je **teorem** ako se može izvesti korištenjem pravila zaključivanja **L** iz praznog skupa formula (bez premisa ili aksioma)

$\vdash_L \omega_i$ formula ω_i je teorem

Primjeri teorema

$$\vdash_L ((P \Rightarrow Q) \wedge (Q \Rightarrow R)) \Rightarrow (P \Rightarrow R)$$

$$\vdash_L Q \Rightarrow (P \Rightarrow (P \Rightarrow (Q \Rightarrow P)))$$

Terminologija formalnih sustava

Konzistentnost (engl. *consistency*)

- Skup formula Γ je **konzistentan** ako i samo ako ne sadrži formule na temelju kojih bi ω_i i $\neg\omega_i$ (istovremeno) bile dedukcije.

Primjeri

- $\Gamma = \{ P, (P \Rightarrow Q) \}$ je konzistentan skup.

- $\Gamma = \{ P, \neg P, (P \Rightarrow Q) \}$ je **nekonzistentan** ili **kontradiktoran** jer su P i $\neg$P istovremeno dedukcije (kontradiktorni aksiomi se nalaze već u samom skupu Γ).

- $\Gamma = \{ P, \neg Q, (P \Rightarrow Q) \}$ je nekonzistentan jer sadrži $\neg$Q, a pravilom "Modus ponens" može se izvesti Q, dakle $\neg$Q i Q bi istovremeno bile dedukcije.

Terminologija formalnih sustava

Odredivost (odlučljivost, engl. *decidability*)

- Neka se u formalnom sustavu {Γ, **L**} izvodi neka formula ω_i (tražimo odgovor je li ω_i dedukcija).

- Formalan sustav je **odrediv** ili **odlučljiv** (engl. *decidable*), ako i samo ako postoji postupak, procedura ili **algoritam** koji će u konačnom vremenu odrediti ili ne dedukciju ω_i (dati u konačnom vremenu odgovor da je ω_i dedukcija ili da ω_i nije dedukcija).
- Formalan sustav {Γ, L} je **poluodrediv** ili **poluodlučljiv** (engl. *semidecidable*), ako i samo postoji algoritam koji će u konačnom vremenu odrediti dedukciju ako ona postoji. Algoritam završava u konačnom vremenu s odgovorom "da" (za dedukciju ω_i), ali ne mora završiti u konačnom vremenu s odgovorom "ne" (ako ω_i nije dedukcija). Moguća je i alternativa (završava za "ne", a ne mora završiti za "da")
- Formalan sustav je **neodrediv** ili **neodlučljiv** (engl. *undecidable*) ako nije ni odrediv ni poluodrediv.

Terminologija formalnih sustava

Semantika u formalnom sustavu povezana je s:

- **interpretacijom** - pridruživanjem istinitosti atomima i
- **evaluacijom** - izračunavanjem istinitosti složene formule.

Model (engl. *model*)

- Neka interpretacija je **model** formalnog sustava {Γ, **L**} ako evaluira **sve** njegove formule u istinito

  Primjer:

  I: {P=T, Q=F, R=F} formule $(Q \vee (((\neg Q) \wedge P) \Rightarrow R))$

  nije model jer ta interpretacija formuli daje neistinitu vrijednost.

Logička zadovoljivost (engl. *logical satisfiability*)

- Skup formula je logički **zadovoljiv** ako ima (barem jedan) model.
- Vrijedi i za pojedinačne formule.
- Sukladno ranijoj definiciji, **logički nezadovoljiv** (**nekonzistentan, kontradiktoran**) skup formula nema nijedan model.

Terminologija formalnih sustava

Logička posljedica (engl. *logical consequence*)

- Formula ω je **logička posljedica** skupa formula Γ, ako je svaki model od Γ ujedno i model od ω
- Kažemo i da skup formula Γ **povlači** (engl. *entails*) formulu ω

- Oznaka logičke posljedice:

  $\Gamma \models \omega$

Valjanost

- Formula je **valjana (engl. *valid*)** ili **tautologija (engl. *tautology*)** ako je istinita **za svaku** interpretaciju i evaluaciju.
- Oznaka valjane formule:

  $\models \omega$ (svaka interpretacija je model formule ω)

Terminologija formalnih sustava

- Logička posljedica izrečena na drugi način: Ako svaka interpretacija koja lijevoj strani od znaka $\models$ daje istinitost ujedno daje i desnoj strani istinitost, tada je desna strana logička posljedica lijeve.

Primjeri logičkih posljedica

- 1. $(P \wedge Q) \models P$

  lijeva strana = T samo za (P=T, Q=T), samo jedan model, a to daje i desnoj strani =T, dakle gornji izraz vrijedi (P je logička posljedica $(P \wedge Q)$).

- 2. $(P \vee Q) \models P$

  lijeva strana je istinita za (P=F, Q=T; P=T, Q=F; P=T, Q=T), ali desna za interpretaciju (P=F, Q=T) nije istinita, te P **nije logička posljedica** $(P \vee Q)$.

- 3. $\{\neg Q, \quad (P \vee Q)\ \} \models P$ (zarez predstavlja konjunkciju $\wedge$)

  skup Γ na lijevoj strani je istinit samo za Q=F, P=T, a to daje istinitost i desnoj strani, te je P logička posljedica navedenog skupa Γ.

- 4. $P \models (Q \vee \neg Q)$

  također vrijedi, jer za svaku interpretaciju za koju je lijeva strana istinita (P=T) i desna stana je istinita (desna strana je doduše uvijek istinita).

Terminologija formalnih sustava

- Skup formula Γ naziva se još **baza znanja** (engl. *knowledge base*, **KB**) formalnog sustava

Primjer: $\Gamma = \{(A \vee C) \wedge (B \vee \neg C)\}$ = baza znanja = dvije konjunkcijom povezane formule (umjesto $\wedge$ može se koristiti zarez).

Neka je: $\alpha = (A \vee B)$

Pitanje: **KB** $\models \alpha$?

DA!

A	B	C	$A \vee C$	$B \vee \neg C$	KB	α
$False$	$False$	$False$	$False$	$True$	$False$	$False$
$False$	$False$	$True$	$True$	$False$	$False$	$False$
$False$	$True$	$False$	$False$	$True$	$False$	$True$
$False$	$True$	$True$	$True$	$True$	$True$	$True$
$True$	$False$	$False$	$True$	$True$	$True$	$True$
$True$	$False$	$True$	$True$	$False$	$False$	$True$
$True$	$True$	$False$	$True$	$True$	$True$	$True$
$True$	$True$	$True$	$True$	$True$	$True$	$True$

Terminologija formalnih sustava

Semantička ekvivalentnost: stroža definicija ekvivalencije dviju formula preko pojma logičke posljedice ($\models$)

- Dvije formule α i β su **semantički ekvivalentne** (oznake ($\alpha \Leftrightarrow \beta$) ili ($\alpha \equiv \beta$)) ako i samo ako vrijede (istinite su) logičke posljedice: **($\alpha \models \beta$) i ($\beta \models \alpha$)**.
- Ranija tablica pravila ekvivalencije daje:

  $(\alpha \Leftrightarrow \beta) = (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
- Ako su α i β ekvivalentne, formula $((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$ mora uvijek biti istinita:

  $\models$ **$((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$**
- <u>Semantička</u> ekvivalencija je na taj način identična <u>dokazljivoj</u> ekvivalenciji: ako želimo dokazati ekvivalentnost dviju formula, dokažemo da je formula **$((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$ tautologija** ili da je njena negacija nezadovoljiva.

Terminologija formalnih sustava

Ispravnost (engl. *soundness*) i kompletnost (engl. *completeness*)

- Formalan sustav {Γ, **L**} je **ispravan** (engl. *sound*) ako je svaka (pravilima dokazana) dedukcija ujedno i logička posljedica skupa formula Γ, tj.:

  $\Gamma \vdash_L \omega_i \quad \Rightarrow \quad \Gamma \models \omega_i$

Neformalno, ispravnost osigurava da je svaka činjenica koja se dokaže istinita.

- Formalan sustav {Γ, **L**} je **kompletan** (engl. *complete*) ako je svaku logičku posljedicu skupa Γ moguće dokazati pravilima L, tj.:

  $\Gamma \models \omega_i \quad \Rightarrow \quad \Gamma \vdash_L \omega_i$

Neformalno, kompletnost osigurava da se mogu dokazati **sve** istinite činjenice.

- Primijetiti: u kompletnom sustavu vrijedi: $\models \omega_i \Rightarrow \vdash_L \omega_i$
- U ispravnom i kompletnom formalnom sustavu {Γ, **L**} vrijedi:
- $\Gamma \models \omega_i \quad = \quad \Gamma \vdash_L \omega_i$ (logička posljedica je ujedno dedukcija i obratno)

Terminologija formalnih sustava

Većina interesantnih formalnih logičkih sustava je nekompletna, a vrlo malo ih je odredivo.

Propozicijska logika je:

- Ispravna, kompletna i odrediva (npr. preslikavanjem u tablicu istinitosti), jer operira s konačnim skupom simbola.

Predikatna logika prvoga reda je:

- Poluodrediva (ako dedukcija postoji, dokazat će se, a ako ne postoji može se ali i ne mora dokazati).
- Odredivi su samo neki podskupovi logike prvog reda.
- "Čista" (npr. bez aritmetike) predikatna logika je ispravna i kompletna (Gödel).

Predikatne logike višega reda nisu kompletne pod pretpostavkom pune semantike.

Normalni oblici propozicijskih formula

- Svaka formula propozicijske logike može se preslikati (ekvivalentna je) formuli u disjunkcijskom normalnom obliku (**DNF**) :

$$(k1_1 \wedge \ldots \wedge k1_n) \vee (k2_1 \wedge \ldots \wedge k2_m) \vee \ldots \vee (kp_1 \wedge \ldots \wedge kp_r)$$

- Svaka propozicijska formula može se preslikati (ekvivalentna je) formuli u konjunkcijskom normalnom obliku (**CNF**) :

$$(k1_1 \vee \ldots \vee k1_n) \wedge (k2_1 \vee \ldots \vee k2_m) \wedge \ldots \wedge (kp_1 \vee \ldots \vee kp_r)$$

gdje su:

- k_i = literal (negirani ili nenegirani atomički simbol - atom)
- **klauzula = disjunkcija literala**. Npr.: $(k2_1 \vee \ldots \vee k2_m)$
- Preslikavanje CNF u DNF i obrnuto je računalno vrlo skupo (vremenski i prostorno). Spada u razred NP-teških (engl. *NP-hard*) problema.

Pretvorba u normalni oblik CNF

Svaka formula u propozicijskoj logici može se pretvoriti u konjunkciju klauzula (CNF):

Npr: $\neg(P \Rightarrow Q) \vee (R \Rightarrow P)$

Algoritam:

1. **Eliminiraj ekvivalencije i implikacije uporabom ekvivalentnog "$\vee$" oblika:**

   $\neg(\neg P \vee Q) \vee (\neg R \vee P)$

2. **Reduciraj doseg negacije (pomak u desno) uporabom DeMorganovih pravila, te eliminiraj dvostruke negacije:**

   $(P \wedge \neg Q) \vee (\neg R \vee P)$

3. **Pretvori u CNF asocijativnim i distribucijskim pravilima:**

   $(P \vee \neg R \vee P) \wedge (\neg Q \vee \neg R \vee P)$,

   te dalje:

   $(P \vee \neg R) \wedge (\neg Q \vee \neg R \vee P)$ **= CNF oblik**

- **Napomena: pretvorba u CNF-oblik ne dovodi nužno do minimalnog CNF-oblika, minimizacija se naknadno provodi prema potrebi.**

SAT-problem – temeljni NP problem

- Traži se model skupa formula Γ (interpretaciju koja evaluira sve formule u skupu Γ u istinito. To je ekvivalentno traženju modela **jedne** složene formule koja se sastoji iz **konjunkcije svih formula u** Γ.
- CNF-oblik skupa formula:

  $(k1_1 \vee \ldots \vee k1_p) \wedge (k2_1 \vee \ldots \vee k2_r) \wedge \ \ldots \wedge (kp_1 \vee \ldots \vee kp_s)$

- Iscrpna procedura rješavanja **CNF** SAT-problema sistematski pridjeljuje istinitosne vrijednosti atomičkim propozicijskim simbolima. **Za *n* atoma 2^n pridruživanja**. Eksponencijalna složenost!

- CNF 2SAT (do 2 literala u klauzuli) - polinomna složenost
- **CNF 3SAT (3 literala u klauzuli) - NP-kompletno**, eksponencijalna složenost

- **Zadovoljivost formule u CNF-obliku s 3 i više literala je NP-kompletno.**
- Mnogi stohastički algoritmi troše eksponencijalno vrijeme u najgorem slučaju, ali polinomno u srednjem (očekivanom). Ti se algoritmi zovu SAT-rješavači (engl. *SAT solver*).
- Najpoznatiji suvremeni SAT-*solveri:* zChaff, miniSAT, SatZ

Teorem dedukcije

- **Teorem:** Formula ψ je **logička posljedica** formule φ, tj. $\varphi \models \psi$, ako i samo ako je formula $(\varphi \Rightarrow \psi)$ tautologija (valjana).

- **Dokaz:** Ako je $(\varphi \Rightarrow \psi)$ tautologija (uvijek istinita), onda iz tablice za implikaciju proizlazi da kada je φ istinit da tada i ψ mora biti istinit (jer bi alternativa vodila na neistinu). To je upravo definicija logičke posljedice.

φ	ψ	$(\varphi \Rightarrow \psi)$
F	F	T
F	T	T
T	F	F
T	T	T

- **Korolar (metoda opovrgavanja):** Budući da $(\varphi \Rightarrow \psi)$ mora biti tautologija, to njena negacija

  $\neg(\varphi \Rightarrow \psi) = \neg(\neg\varphi \vee \psi) = (\varphi \wedge \neg\psi)$ mora biti nezadovoljiva. Dakle:

$\varphi \models \psi$ **akko je** $(\varphi \wedge \neg\psi)$ **<u>nezadovoljiva</u>**

Primjer rasuđivanja opovrgavanjem

Neka istinite formule predstavljaju skup Γ :

1. P
2. $(P \Rightarrow Q)$
3. $(Q \Rightarrow S)$

U CNF obliku: $\Gamma = [\ (P)\ \wedge\ (\neg P \vee Q)\ \wedge\ (\neg Q \vee S)\ ]$

- Pitamo se: je li S logička posljedica skupa Γ : $\Gamma \models S$?

- **Teorem dedukcije i korolar: S je logička posljedica Γ ako je $(\Gamma\ \wedge\ \neg S)$ nezadovoljiva.**

- Dakle, skupu Γ dodajemo negaciju formule koju želimo dokazati $(\neg S)$:

  $[\ (P)\ \wedge (\neg P \vee Q)\ \wedge\ (\neg Q \vee S)\ \wedge (\neg S)\ ]$

  Sad možemo iskoristiti npr. SAT-rješavač da pokušamo naći barem jedan model (zadovoljivost). Ako SAT-rješavač pokaže da formulu **nije** moguće zadovoljiti (opovrgnuo ju je – nema modela), zaključujemo:

  S je doista logička posljedica skupa Γ.

Zadaci

1. Dokažite istinitost sljedećih logičkih zaključaka:

   a) $(P \wedge Q) \wedge R, S \wedge T \vdash_L Q \wedge S$

   b) $P \wedge Q \Rightarrow R \vdash_L P \Rightarrow (Q \Rightarrow R)$

   c) $P \Rightarrow Q \vdash_L \neg P \vee Q$

2. Pretvorite propozicijsku formulu u CNF-oblik

   $(P \Rightarrow (Q \Rightarrow R)) \Leftrightarrow (P \Rightarrow (R \Rightarrow Q))$

3. Koristeći teorem o dedukciji, ekvivalencije koje vrijede u propozicijskoj logici i pravila zaključivanja, pokažite da je formula S logička posljedica skupa formula:

   1. P
   2. $(P \Rightarrow Q)$
   3. $(Q \Rightarrow S)$

4. Koristeći pojam logičke posljedice, pokažite ili opovrgnite semantičku ekvivalentnost formula

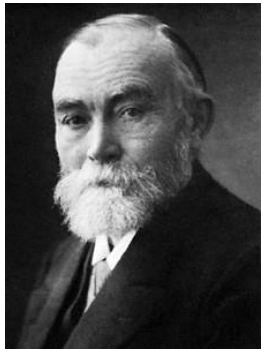
   φ_1: $P \Rightarrow (Q \vee R)$ i

   φ_2: $(\neg Q \wedge \neg R) \Rightarrow P$

Predikatna logika
(Logika predikata prvoga reda)

Engl. *predicate logic*, *predicate calculus*, *first order predicate logic* – FOPL

Gottlob Frege



Kurt Gödel

Alfred Tarski

Predikatna logika – sintaksa

1. P: Svi ljudi su smrtni.
2. Q: Sokrat je čovjek.
3. R: Sokrat je smrtan.

- U propozicijskoj logici nikako se iz 1:P i 2:Q ne može zaključiti 3:R.
  FOPL uvodi objekte, relacije, obilježja, funkcije (**za pobliži opis izjave**).
  Povećana je izražajna moć formalne logike.

Sintaksa:

Atomički predikat:

(pred_simb t1 t2 … tn) – infiks notacija (LISP)
pred_simb(t1 t2 … tn) – prefiks notacija (Prolog)

} oba načina pisanja OK, ali ne miješati

- **pred_simbol:** osnovno obilježje u rečenici (predikat)
- **t_i = članovi:** objekti ili odnosi u rečenici

Predikatna logika – sintaksa

Članovi (t_i) :

- Konstante: objekti u nekom svijetu (blok1, sokrat, …).
- Rezervirane konstante: T, F.
- Varijable: razred objekata ili obilježja;
  mogu poprimiti vrijednosti iz svoje domene; (Npr.: X, Y, …).
- Funkcije: veza između objekata - (fun_simb t1 t2 … tn). Npr.: (cos X), (otac_od abel kain)

Formalna def. člana:

- 1. Konstanta je član.
- 2. Varijabla je član.
- 3. Ako je fun_simb funkcijski simbol sa n-argumenata, a $t_1, t_2, \ldots, t_n$ su članovi, tada je (fun_simb t_1 t_2 … t_n) član.

Logički operatori (vezice): $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$

Kvantifikacijski simboli (uz varijable, pobliže određuju istinitost rečenice):

- $\exists$ (postoji, za_neki, exist) - egzistencijski ili partikularni kvantifikator (barem jedan).
- $\forall$ (za_svaki, svi, for_all) - univerzalni kvantifikator (svi), ima središnju ulogu u izražavanju generalizacije.

Predikatna logika – sintaksa

Ispravno definiran složeni predikat ili formula (wff):

1. svaki atomički predikat je formula.

2. ako je S_i formula, tada su formule:
   $(\neg S)$, $(S_1 \wedge S_2)$, $(S_1 \vee S_2)$, $(S_1 \Rightarrow S_2)$, $(S_1 \Leftrightarrow S_2)$.

3. ako je X varijabla, a S formula, tada su formule: $\exists X\ S(X)$, $\forall X\ S(X)$.
   oznaka S(X) = formula S u kojoj postoji varijabla X

- Negirani ili nenegirani atomički predikat naziva se **literal**.

Primjer ispravno definirane složene formule u infiks notaciji:

$(\forall X\ \forall Y\ (((\text{otac}\ X\ Y) \vee (\text{majka}\ X\ Y)) \Rightarrow (\text{roditelj}\ X\ Y)))$

Predikatna logika – semantika

Skup wff uvijek se odnosi na neku domenu razmatranja D.

Interpretacija (I) je proces preslikavanja elemenata iz domene D svakoj pojedinoj konstanti, varijabli, i funkciji, te atomičkom predikatu, tako da:

- Simbolu T uvijek je pridružena istinita vrijednost.
- Simbolu F uvijek je pridružena neistinita vrijednost.
- Svakoj konstanti pridruži se jedan element iz D.
- Svakom funkcijskom simbolu pridruži se jedan element iz D.
- Svakoj varijabli se pridruži neprazan podskup iz D (dozvoljene supstitucije).

- Svaka funkcija *f*, sa *m* argumenata, definira **interpretacijom i evaluacijom** preslikavanje iz D^m u D, tj. $f: D^m \rightarrow D$ (funkcija se evaluira u jedan element iz D).
- Svaki predikat P, s brojem članova *n*, definira interpretacijom i evaluacijom svojih članova preslikavanje iz D^n u $\{T, F\}$, tj. $P: D^n \rightarrow \{T, F\}$ (predikat se za određene elemente domene preslika u istinu ili laž).
- Vrijednosti wff formula složenih logičkim operatorima dane su odgovarajućim istinitosnim tablicama.
- Vrijednost $\forall X\ P(X)$ je T, ako P(X) je T, za sve vrijednosti X dane sa **I**, a F inače.
- Vrijednost $\exists X\ P(X)$ je T, ako P(X) je T, barem za jednu vrijednost X danoj sa **I**, a F inače.

Predikatna logika – semantika

- Određivanje istinitosti wff svodi se na **interpretaciju + evaluaciju**
- Skup svih istinitih predikata nad domenom D naziva se **stanje svijeta** (*engl. state of the world*).

Primjeri pridruživanja istinitosti:

1. (prijatelj ivan ana)

   predikat je T, ako u D postoji objekt Ana koja je prijatelj Ivanu.

2. Neka je domena od X skup prirodnih brojeva. Tada:

   ∀X (veci X 10) atomički predikat je F

   ∃X (veci X 10) atomički prediakt je T

- ∀ - u određivanju T potrebne sve supstitucije varijable (problem ako je domena beskonačna)
- ∃ - u određivanju T potrebna jedna supstitucija za koju T (problem ako je domena beskonačna i predikat F)

Dopuna pravilima ekvivalencije

Simbol varijable i doseg kvantifikatora:

Neka su P(X), Q(X) wff s varijablom X, tada vrijedi:

$\exists X\ P(X) = \exists Y\ P(Y)$ - simbol varijable nije bitan, ali je bitan

$\forall X\ P(X) = \forall Y\ P(Y)$ doseg, uvijek unutar jedne formule

Proširenje De Morganovih relacija:

Primjer: *"Ne vole svi ići zubaru." = "Postoji netko tko ne voli ići zubaru."*

$\neg(\forall X\ P(X)) = \exists X\ (\neg P(X))$

$\neg(\exists X\ Q(X)) = \forall X\ (\neg Q(X))$

- **Negacija mijenja kvantifikator!**

Dopuna pravilima ekvivalencije

Supstitucija

- Neka je **P(X)** wff s varijablom X.
- Neka je domena X: **D = {1, 2, 3}**
- Formula $\forall X\ P(X)$ je ekvivalentna $[\ P(1) \wedge P(2) \wedge P(3)\ ]$
  $\forall X\ P(X) \equiv [\ P(1) \wedge P(2) \wedge P(3)\ ]$
- $\forall X\ P(X)$ je istinita ako su istinite **sve** supstitucije iz domene.

- Formula $\exists X\ P(X)$ je ekvivalentna $[\ P(1) \vee P(2) \vee P(3)\ ]$
  $\exists X\ P(X) \equiv [\ P(1) \vee P(2) \vee P(3)\ ]$
- $\exists X\ P(X)$ je istinita ako je istinita **barem jedna** supstitucija iz domene.

Permutacija kvantifikatora

- Formule :
  $\forall X\ \exists Y\ P(X,Y) \neq \exists Y\ \forall X\ P(X,Y)$ - nisu ekvivalentne!
  Npr. $\forall x \exists y Voli(x, y)$: svatko voli nekoga i $\exists y \forall x Voli(x, y)$: postoji netko koga svi vole

Ispravna uporaba univerzalnog kvantifikatora $\forall$

Primjer:

- Neka je okvir razmatranja (skup objekata): {Garfield, Feliks, računalo}
- Preslikaj u predikatnu logiku rečenicu: "Sve mačke su sisavci."
- **Za sve** objekte x u okviru razmatranja vrijedi: ako su mačke tada su sisavci.

  $\forall x$ [mačka(x) $\Rightarrow$ sisavac(x)]
- Dokaz: Supstitucija svih objekata u formulu (konjunkcija formula jer $\forall$):

  [mačka(Garfield) $\Rightarrow$ sisavac(Garfield)] $\wedge$ [mačka(Feliks) $\Rightarrow$ sisavac(Feliks)] $\wedge$ [mačka(računalo) $\Rightarrow$ sisavac(računalo)]

- prva [T $\Rightarrow$ T]: T (vidi tablicu za $\Rightarrow$)
- druga [T $\Rightarrow$ T]: T (vidi tablicu za $\Rightarrow$)
- treća [F $\Rightarrow$ F]: T (vidi tablicu za $\Rightarrow$)

  time je ukupna formula = T !!!

Ispravna uporaba univerzalnog kvantifikatora ∀

Primjer (nastavak):

- Da smo napisali: $\forall x$ [mačka(x) $\wedge$ sisavac(x)]

Doslovno: "svaki x je mačka i svaki x je sisavac"

- Supstitucija svih objekata u tom slučaju daje:
  [mačka(Garfield) $\wedge$ sisavac(Garfield)] $\wedge$ [mačka(Feliks) $\wedge$ sisavac(Feliks)] $\wedge$
  [mačka(računalo) $\wedge$ sisavac(računalo)]

- mačka(računalo) = F - **daje neistinitu cijelu formulu !!!**

Ispravna uporaba egzistencijskog kvantifikatora ∃

Primjer:

- Neka je okvir razmatranja (kao i prije): {Garfield, Feliks, računalo}
- Preslikaj u predikatnu logiku: "Garfield ima brata koji je mačka."

- Postoji **barem jedan** (neki) objekt i takav da su mu obilježja istinita.
  $\exists x$ [brat(x, Garfield) $\wedge$ mačka(x)]

- Dokaz supstitucijom svih objekata u formulu (disjunkcija formula jer ∃):

  [brat(Garfield, Garfield) $\wedge$ mačka(Garfield)] $\vee$
  [brat(Feliks, Garfield) $\wedge$ mačka(Feliks)] $\vee$
  [brat(računalo, Garfield) $\wedge$ mačka(računalo)]

- Prva [ ] neistinita jer Garfield nije sam sebi brat, ali idemo dalje jer su […] povezane disjunkcijom.
- Drugi red istinit, cijela formula je istinita (dalje ne moramo ispitivati).

Ispravna uporaba egzistencijskog kvantifikatora ∃

Primjer (nastavak):

- Ako bi preslikali: $\exists x$ [brat(x, Garfield) $\Rightarrow$ mačka(x)]
- To se drugačije može napisati kao: $\exists x$ [$\neg$brat(x, Garfield) $\vee$ mačka(x)] – doslovno: "postoji takav x koji ili nije brat od Garfielda ili je mačka"

- Supstitucija svih objekata u disjunkciju formula daje:

  [brat(Garfield, Garfield) $\Rightarrow$ mačka(Garfield)] $\vee$

  [brat(Feliks, Garfield) $\Rightarrow$ mačka(Feliks)] $\vee$

  [brat(računalo, Garfield) $\Rightarrow$ mačka(računalo)]

- Implikacija je istinita ako je atomički izraz na lijevoj strani neistinit!
- Npr. ako je: [brat(računalo, Garfield) $\Rightarrow$ mačka(računalo)] istinito, cijela je formula istinita !

- Egzistencijski kvantificirana implikacijska formula je istinita ako u okviru razmatranja postoji barem jedan objekt za koji je premisa implikacije **neistinita** (desna strana može biti T ili F).
- Takva rečenica **ne daje nikakvu potvrdnu informaciju**.

- Zaključak: $\forall$ ide uz $\Rightarrow$

  $\exists$ ide uz $\wedge$

Obilježja predikatne logike

- Zadovoljivost
- Model
- Logička posljedica
- Kontradiktornost
- Pravila zaključivanja

Sva navedena svojstva su jednaka kao i u propozicijskoj logici.

- **Predikatna logika višega reda (engl. *Higher-Order Logic*):**
  - Kvantifikacija na predikatnom (ili funkcijskom) simbolu.

Npr: $\forall$(Voli) (Voli ivo ana)

Zadaci

1. Preslikajte sljedeće rečenice prirodnog jezika u formalizam predikatne logike prvoga reda (FOPL). Pritom definirajte sve potrebne predikate i konstante.

   a) „Niti jedan student ne sluša sve predmete.”

   b) „Svaki profesor je zaposlenik samo jednog fakulteta, a predaje na jednom ili više fakulteta.”

   c) „Svatko voli nekog i nitko ne voli svakog.”

   d) „Neki studenti koji slušaju predmet FMuOS također slušaju i predmet NOS.”

Dodatak: bitna pravila prirodnog zaključivanja u propozicijskoj logici

The basic rules of natural deduction:

	introduction	*elimination*
$\wedge$	$\dfrac{\phi \quad \psi}{\phi \wedge \psi}\,\wedge\mathrm{i}$	$\dfrac{\phi \wedge \psi}{\phi}\,\wedge\mathrm{e}_1 \qquad \dfrac{\phi \wedge \psi}{\psi}\,\wedge\mathrm{e}_2$
$\vee$	$\dfrac{\phi}{\phi \vee \psi}\,\vee\mathrm{i}_1 \qquad \dfrac{\psi}{\phi \vee \psi}\,\vee\mathrm{i}_2$	$\dfrac{\phi \vee \psi \quad \boxed{\begin{array}{c}\phi\\ \vdots\\ \chi\end{array}} \quad \boxed{\begin{array}{c}\psi\\ \vdots\\ \chi\end{array}}}{\chi}\,\vee\mathrm{e}$
$\rightarrow$	$\dfrac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \psi\end{array}}}{\phi \rightarrow \psi}\,\rightarrow\mathrm{i}$	$\dfrac{\phi \quad \phi \rightarrow \psi}{\psi}\,\rightarrow\mathrm{e}$
$\neg$	$\dfrac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \bot\end{array}}}{\neg\phi}\,\neg\mathrm{i}$	$\dfrac{\phi \quad \neg\phi}{\bot}\,\neg\mathrm{e}$
$\bot$	(no introduction rule for $\bot$)	$\dfrac{\bot}{\phi}\,\bot\mathrm{e}$
$\neg\neg$		$\dfrac{\neg\neg\phi}{\phi}\,\neg\neg\mathrm{e}$

Some useful derived rules:

$$\dfrac{\phi \rightarrow \psi \quad \neg\psi}{\neg\phi}\,\mathrm{MT} \qquad \dfrac{\phi}{\neg\neg\phi}\,\neg\neg\mathrm{i}$$

$$\dfrac{\boxed{\begin{array}{c}\neg\phi\\ \vdots\\ \bot\end{array}}}{\phi}\,\mathrm{PBC} \qquad \dfrac{}{\phi \vee \neg\phi}\,\mathrm{LEM}$$

- Izvor: Huth, Ryan, **Logic in Computer Science, Cambridge University Press, 2004.**

- Objašnjenja nekih pojmova:

  $\rightarrow$e = modus ponens

  MT = modus tollens

  PBC (Proof By Contradiction)

  LEM (Law of the Excluded Middle) = TND (Tertium Non Datur) = Trećega nema

FVPP: Teorijske osnove formalne verifikacije provjerom modela

Vremenska logika CTL

Formalna verifikacija kritičnih programskih dijelova u sustavu NuSMV

Pripremio: izv. prof. dr. sc. Alan Jović

Ak. god. 2022./2023.

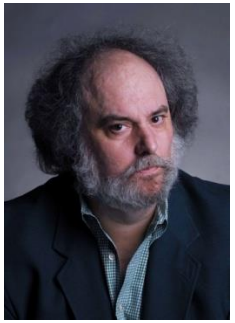
Vremenska logika s grananjem

Engl. *Computational Tree Logic*, **CTL**

M. Ben-Ari

E. M. Clarke

E. A. Emerson



Modalna i vremenska logika

- **Modalna logika** - proširenje klasične logike „modalitetima" istinitosti (subjektivnim konceptima), kao što su: „što mora biti istinito" i „što može biti istinito."

Primjer:

- p = atomički propozicijski simbol
- Neka je: p = F (neisitinit) u sadašnjem svijetu (stanju stvari).

- Tada:
- (*moguće* p) = T — ako postoji barem jedan drugi svijet (neka druga situacija, neki drugi scenarij, neka druga baza znanja) u kojoj je p = T.
- (*nužno* p) = F — jer (*nužno* p) = T akko je p istinit u svim svjetovima (ovdje F jer je u našem svijetu p=F).

- U klasičnu propozicijsku i predikatnu logiku dodaju se modalni operatori.
- Modalni opetarori u vremenskoj logici: uvijek, konačno, što_je_bilo, što_je_sada, što_će_biti

Vremenska logika – višestruki pogledi

- **TL propozicijska ili predikatna:** Klasična propozicijska logika proširena vremenskim operatorima. TL prvoga reda nije odlučljiva, ali neki njezine reducirane varijante jesu odlučljive (npr. one do najviše jedne slobodne varijable u predikatnoj formuli, za formule s predikatima od 1 ili 2 varijable).
- **Globalna ili modularna** (endogena ili egzogena). Rasuđivanje o kompletnom sustavu korištenjem vremenskih formula – omogućuje konkurentnost izvođenja, ili samo ograničenje na pojedine dijelove.
- **TL linearnog vremena ili s grananjem vremena:** U svakom trenutku postoji samo jedan budući trenutak (jedna vremenska crta) ili u svakom trenutku može postojati više različitih budućih vremenskih crta.
- **Diskretno ili kontinuirano vrijeme.** U računarstvu uobičajeno diskretno vrijeme (sekvence stanja).
- **Uključuje ili ne uključuje prošlo vrijeme, uz buduće vrijeme.** Izvorno TL obuhvaća oba vremena. U digitalnim sustavima uobičajeni su samo operatori budućeg vremena.

- Trenutačno razmatramo: **TL propozicijsku, globalnu (endogenu), s grananjem vremena, diskretno vrijeme, samo buduće vrijeme: CTL**

Kripkeova struktura

Model sustava koji želimo formalno verificirati :

= model implementacije (***I***)

= Kripkeova struktura **M** (Saul Kripke, 1971)

$$I = M = (S, R, L)$$

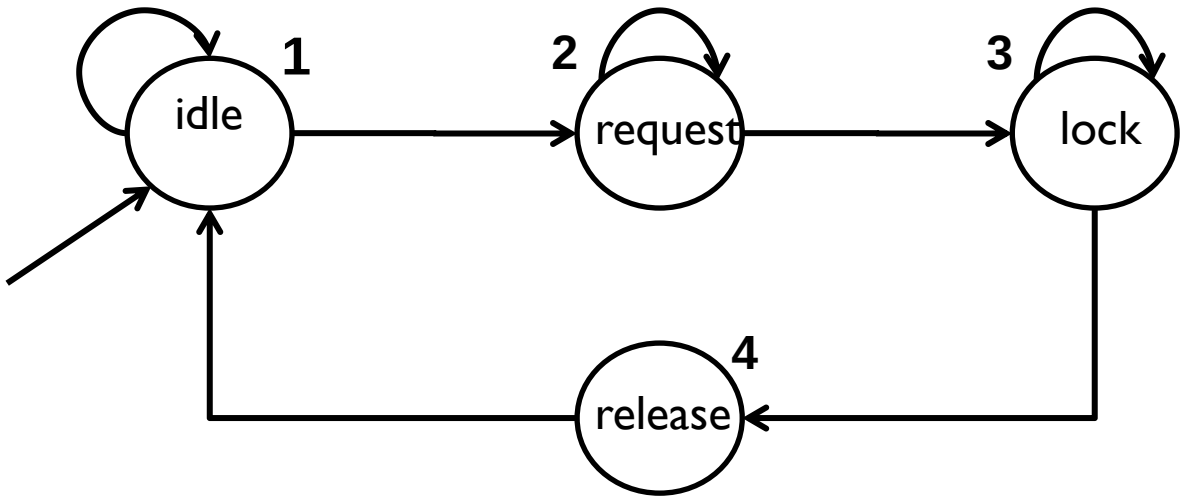
gdje je :

- **S** : konačan skup stanja
- $\mathbf{R} \subseteq \mathbf{S} \times \mathbf{S}$: totalna binarna relacija (svako stanje je obuhvaćeno; ima sljedbenika), tj. $\forall s \in \mathbf{S} \ (\exists t \in \mathbf{S} \mid (s, t) \in \mathbf{R})$
- **L** : funkcija označavanja stanja: $\mathbf{S} \rightarrow 2^{\mathbf{AP}}$

  AP: skup atomičkih propozicijskih simbola

  L: (engl. *labeling*) daje interpretaciju svih simbola iz skupa AP za svako stanje $s \in \mathbf{S}$.

Primjer Kripkeove strukture

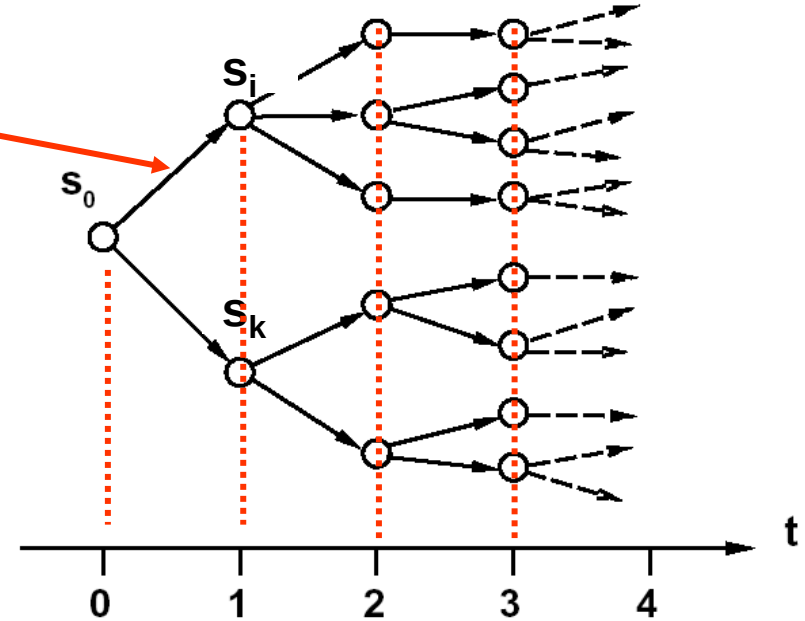


- **S= {1, 2, 3, 4}**
- **R = {(1,1), (1,2), (2,2), (2,3), (3,3), (3,4), (4,1)}**
- **L: L(1) = {idle}**
  L(2) = {request}
  L(3) = {lock}
  L(4) = {release}
- **Jedan put** (engl. *path*), ili **jedno odvijanje izvođenja sustava** (engl. *one execution, one computation*) je beskonačna sekvenca (slijed) stanja.
- Npr.: 1, 2, 2, 2, 2, 3, 4, 1, 1, …
- Mogući prijelazi stanja obuhvaćeni su relacijom R.

Vremenska logika s grananjem

- Neka definirana Kripkeova struktura može se promatrati kao beskonačno stablo izvođenja sustava ("odmota" se počevši od promatranog stanja s_0)
- **To je vremenska logika s grananjem** (engl. *Computation Tree Logic*, CTL)
- Razvijena početkom 1980-tih, koristi se naveliko i danas (razvili: Ben-Ari, Clarke, Emerson i drugi)

mogući prijelazi



Pokazuje **sve** putove izvođenja

Formula CTL-a (u kontekstu odmotane Kripkeove strukture)

- Kvantifikator puta **Q** i operator stanja **L** čine par **Q L** , koji poprima 8 dozvoljenih kombinacija:

Q L

- **X** immediate successor
- **G** for all states
- **F** there exists a state
- **U** from this state until some state

- **E** there exists a path
- **A** for all paths

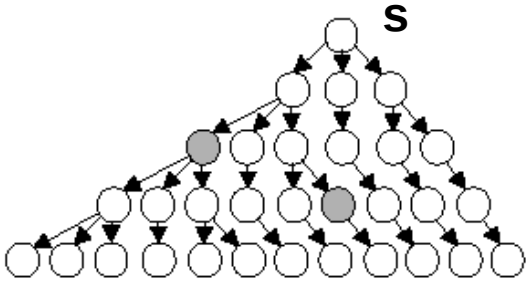
PAZI: Uvijek u paru !

EX	AX
EG	AG
EF	AF
EU	AU

Semantika formule u CTL-u za model **M** i stanje **S**

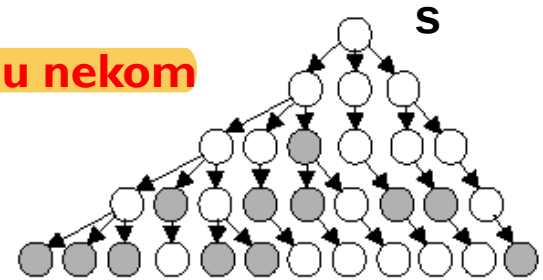
CTL Formulas

EF p "there is a path such that p eventually holds"



EF p = "postoji put (barem jedan) na kojem u nekom budućem stanju vrijedi p"

AF p "all paths are such that p eventually holds"



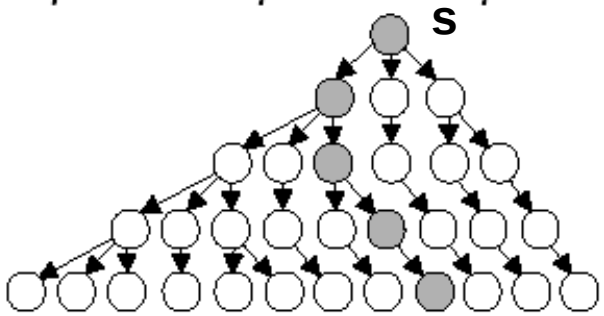
AF p = "na svim putovima u nekom budućem stanju vrijedi p"

Napomena: EF p i AF p su istinite ako već i u promatranom (početnom) stanju vrijedi p.

Semantika formule u CTL-u za model **M** i stanje **S**

CTL Formulas Cont.

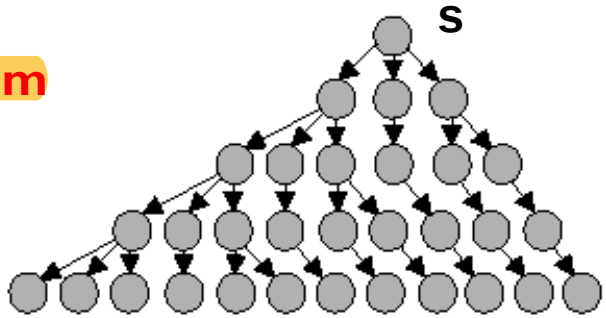
EG p "there is a path such that p holds invariantly along the path"



EG p = "postoji put (barem jedan) na kojem u svim budućim stanjima vrijedi p"

AG p "every path is such that p holds invariantly along the path"

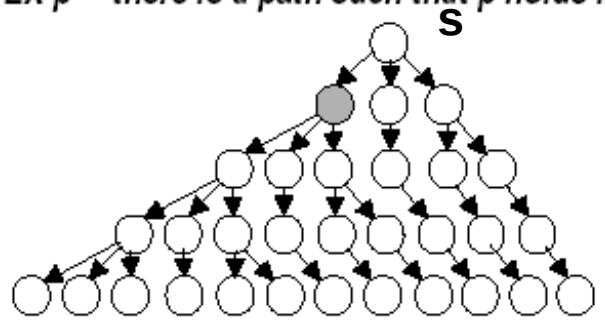
AG p = "na svim putovima u svim budućim stanjima vrijedi p"



Semantika formule u CTL-u za model **M** i stanje **S**

CTL Formulas Cont.

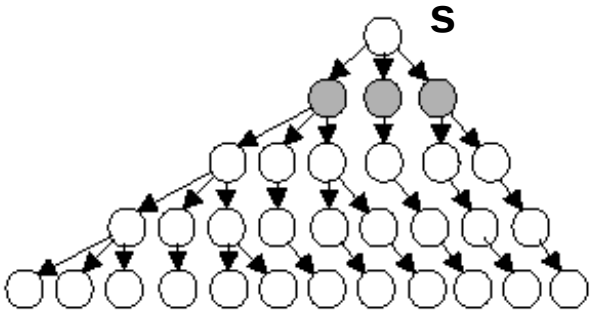
EX p "there is a path such that p holds in the next state along the path"



EX p = "postoji put (barem jedan) na kojem u sljedećem stanju vrijedi p"

AX p "every path is such that p holds in the next state along the path"

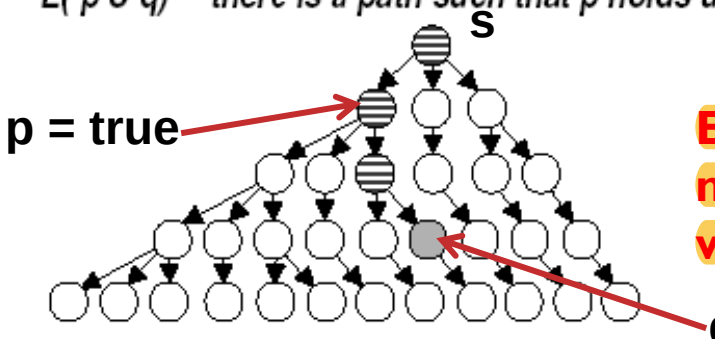
AX p = "na svim putovima u sljedećem stanju vrijedi p"



Semantika formule u CTL-u za model **M** i stanje **S**

CTL Formulas Cont.

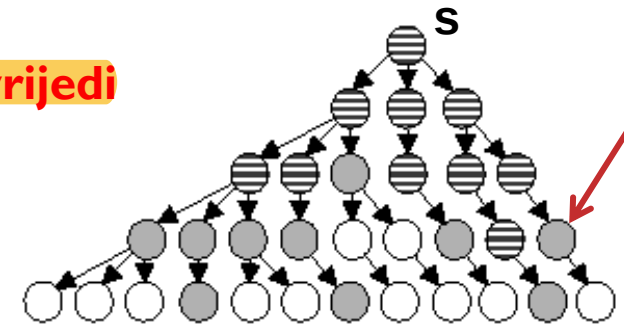
E(p U q) *"there is a path such that p holds until q holds"*



E (p U q) = "postoji put (barem jedan) na kojem vrijedi p sve dok ne počne vrijediti q"

Ovdje p može i ne mora biti istinit

A (p U q) *"every path is such that p holds until q holds"*



A (p U q) = "na svim putovima vrijedi p sve dok ne počne vrijediti q"

Napomena: E(p U q) i A(p U q) su istinite ako već i u promatranom (početnom) stanju vrijedi q (p ne mora uopće vrijediti).

CTL – sintaksa

- AU, EU - binarni operatori
- Ostalih 6 operatora - unarni operatori

Neke ispravno definirane CTL-formule:

- AG $(q \Rightarrow EG\ r)$
- EG p
- $p \wedge q$
- E (p U q)
- A (p U EF p)
- AG $(p \Rightarrow A\ [p\ U\ (\neg p \wedge A\ [\neg p\ U\ q]\ )\ ]\ )$

Neke krivo definirane CTL-formule:

- FG p
- EF (r U q)
- A[p U (q U r)]
- AF $[(r\ U\ q) \wedge (p\ U\ r)]$

CTL – semantika

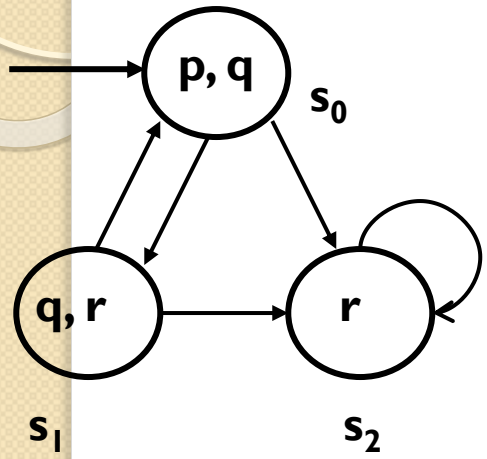
M = (S, R, L) - model sustava (Kripkeova struktura)

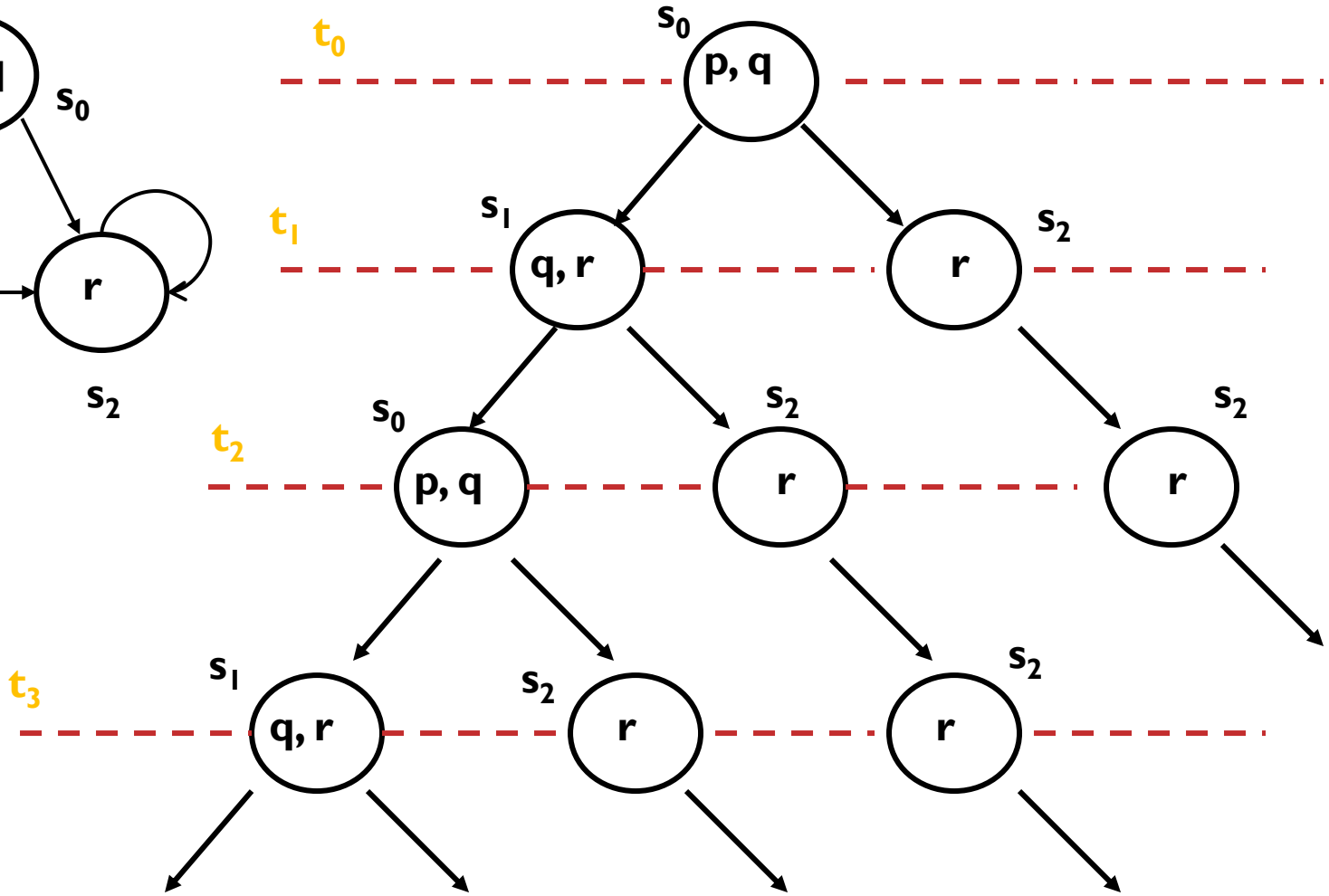
- $M, s \models \varphi$ formula vrem. logike φ je istinita u modelu M za stanje s
- $M, s \not\models \varphi$ formula vrem. logike φ nije istinita u modelu M za stanje s

1. $M, s \models p$	istinita	akko $p \in L(s)$ (p je propozicijski atomički simbol)
2. $M, s \models (\varphi_1 \wedge \varphi_2)$		akko $M, s \models \varphi_1$ i $M, s \models \varphi_2$
3. $M, s \models (\varphi_1 \vee \varphi_2)$		akko $M, s \models \varphi_1$ ili $M, s \models \varphi_2$
4. $M, s \models (\varphi_1 \Rightarrow \varphi_2)$		akko $M, s \not\models \varphi_1$ ili $M, s \models \varphi_2$ (implikacija)
5. $M, s \models AX\ \varphi$		akko za sve s_i takve da $s \rightarrow s_i$ vrijedi $M, s_i \models \varphi$

Analogne formalne definicije i za $EX\ \varphi$, $EF\ \varphi$, $AF\ \varphi$, $EG\ \varphi$, $AG\ \varphi$, $E(\varphi_1 U \varphi_2)$ i $A(\varphi_1 U \varphi_2)$

Primjer "odmotavanja" modela





- p, q, r - propozicijski simboli
- S_0 - početno stanje ili stanje koje nas zanima

Neke istinite CTL-formule za model s prethodnog slajda

- $M, s_0 \models (p \wedge q)$; atomi p i q su istiniti u stanju s_0
- $M, s_0 \models \neg r$; atom r nije istinit u stanju s_0
- $M, s_0 \models EX\ (q \wedge r)$; postoji put gdje je za sljedeće stanje vrijedi $(q \wedge r)$
- $M, s_0 \models \neg AX\ (q \wedge r)$; postoji jedan put na kojem ne vrijedi za sljedeće stanje $(q \wedge r)$
- $M, s_0 \models \neg EF\ (p \wedge r)$; nema puta sa stanjem za koje vrijedi $(p \wedge r)$
- $M, s_0 \models AF\ r$; duž svih putova možemo dosegnuti stanje za koje vrijedi r
- $M, s_0 \models E\ [(p \wedge q)\ U\ r]$; postoji put iz s_0 na kojem u svim stanjima $(p \wedge q)$ = True, dok r=True (npr. do s_2)
- $M, s_0 \models A\ [p\ U\ r]$; na svim putovima vrijedi [p U r]

Ekvivalentne CTL-formule i adekvatni skup

$\neg$**AF** φ	=	**EG** $\neg\varphi$	**; de Morgan**
AF φ	=	$\neg$**EG** $\neg\varphi$	
EG φ	=	$\neg$**AF** $\neg\varphi$	
AG $\neg\varphi$	=	$\neg$**EF** φ	**; de Morgan**
AG φ	=	$\neg$**EF** $\neg\varphi$	
$\neg$**AX** φ	=	**EX** $\neg\varphi$	**; X je vlastiti dual**
AX φ	=	$\neg$**EX** $\neg\varphi$	
AF φ	=	**A (True U** φ**)** = $\neg$**EG** $\neg\varphi$	
EF φ	=	**E (True U** φ**)**	
EG φ	=	$\neg$**[A [True U** $\neg\varphi$**]]**	

- Ponekad notacija: A[p U q] = [p AU q]
  E[p U q] = [p EU q]

- Temeljem gornjih ekvivalencija, za izračun svih CTL-formula dovoljno je imati postupke za izračun EX, EG, EU = **adekvatni skup** (engl. *adequate set*). Postoji više takvih adekvatnih skupova.

Primjeri preslikavanja prirodnog jezika u CTL-formule – specifikacija ponašanja

1. „Moguće je doći u stanje gdje start=T i ready=F.”

EF (start $\wedge$ $\neg$ready)

2. „Za svako stanje, ako se postavi zahtjev (npr. za nekim resursom), onda će on biti konačno prihvaćen (kad-tad).”

AG (zahtjev $\Rightarrow$ AF prihvaćen)

3. „U svakom slučaju, određeni proces će nakon konačnog vremena biti stalno zaustavljen.”

AF (AG zaustavljen)

4. „Iz svakog stanja moguće je barem na nekom putu doći do stanja "restart".”

AG (EF restart)

Primjeri preslikavanja prirodnog jezika u CTL-formule – specifikacija ponašanja

5. „Kadgod je in = 1, nakon dva takta je uvijek out = 1.”

AG (in = 1 $\Rightarrow$ AX AX out = 1)

6. „Uvijek vrijedi: ako se pojavi "send" onda konačno "receive” postaje istinit, te do tog trenutka "send" mora ostati istinit.”

AG (send $\Rightarrow$ A(send U receive))

Primjena CTL-a u provjeri modela (engl. *CTL model checking*)

- Za danu Kripkeovu strukturu (usmjereni označeni graf) i određen skup početnih stanja S_0, provjeri da CTL-formula vrijedi za ta stanja.
- Formalno: $M, S_0 \models \phi$, tj. $\forall s_0 \in S_0 \quad M, s_0 \models \phi$ (za svako stanje iz S_0)

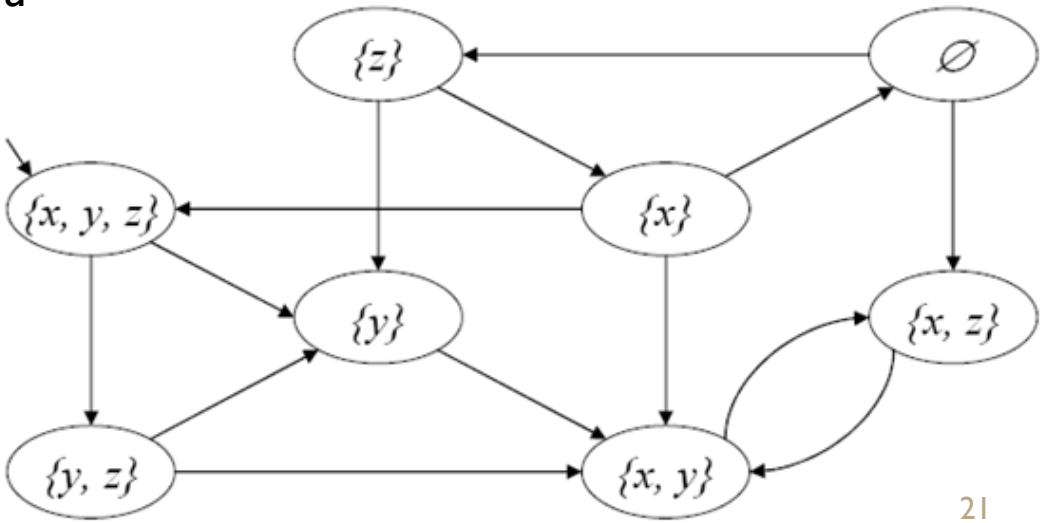
Postupak

- Potrebno je pronaći sva stanja koja zadovoljavaju CTL-formulu ϕ i ispitati da li je željeni podskup S_0 uključen.
- Provjera modela u CTL-u povlači **rukovanje skupovima stanja**.

Rješavanje ugniježđenih CTL- formula

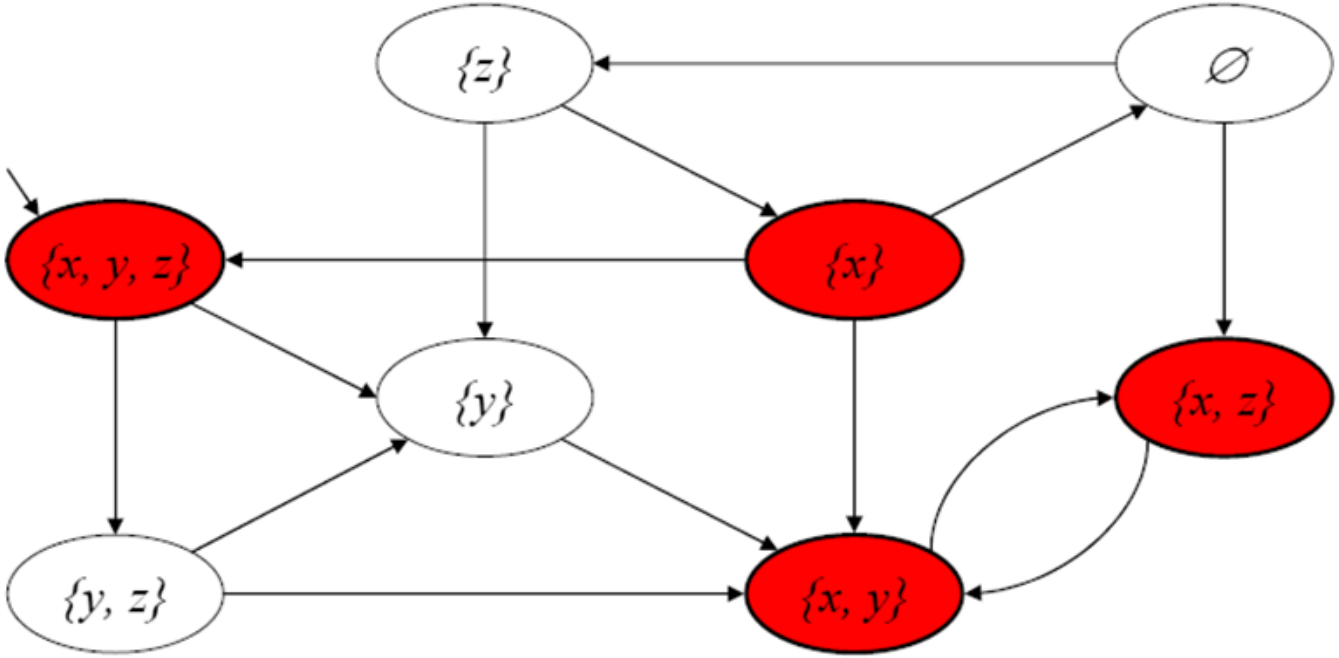
- Primjenjuje se induktivno pravilom "iznutra prema van"
- Računa se skup stanja koja zadovoljavaju najviše unutarnju CTL-formulu
- Koriste se ti rezultati za izračun sve više vanjskih formula

Primjer (UNESCO math&dev TUNIS 2008):
Za zadanu Kripkeovu strukturu izračunaj skup stanja S_k koja zadovoljavaju CTL formulu AF AG(x)



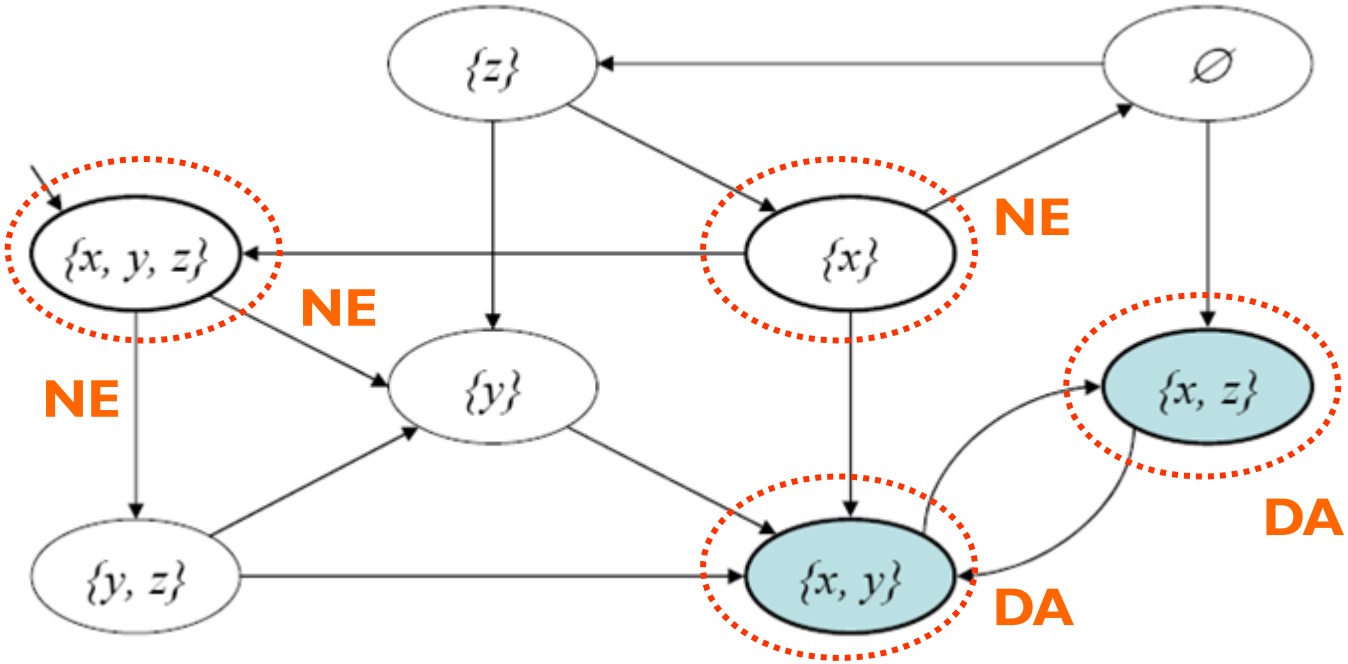
Postupak:
1. Izračunaj $S_k(x)$
2. Izračunaj $S_k(AG(x))$
3. Izračunaj $S_k(AF(AG(x)))$

Skup stanja koja zadovoljavaju formulu x = $S_k(x)$:



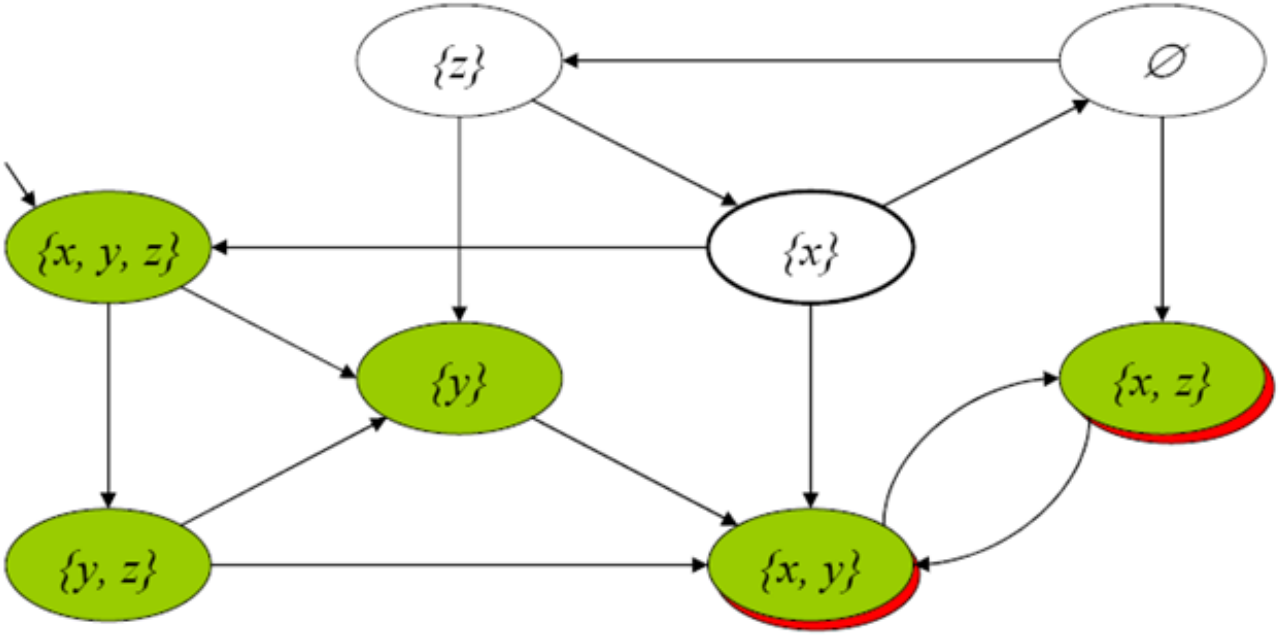
Skup stanja koja zadovoljavaju formulu AG(x) = S_k(AG(x)):

- Da bi se zadržalo, stanje koje sadrži x mora biti korijen prijelaza koja dalje sva sadržavaju x – trebaju se analizirati prethodnici stanja u kojima vrijedi x



Skup stanja koja zadovoljavaju formulu AF(AG(x)) = S_k(AF(AG(x))):

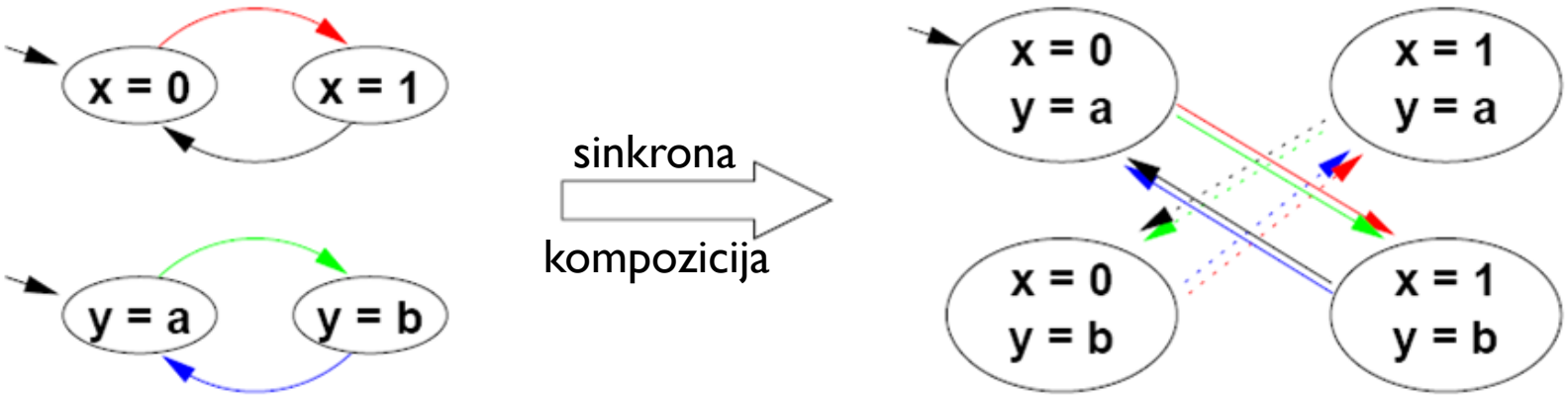
- Tražimo prethodnike prošla dva stanja (koja zadovoljavaju AG(x)) koji na svim svojim putevima konačno dolaze do ta dva stanja, uklone se oni prethodnici za koje to ne vrijedi.



Sinkrona i asinkrona kompozicija Kripkeovih struktura

- Složeni Kripkeovi modeli tipično se grade od jednostavnih.
- Sinkrona kompozicija: **u svakom vremenskom trenutku sve komponente mijenjaju stanje** (istodobno, paralelno).
- Tipičan primjer: **sekvencijsko** sklopovlje

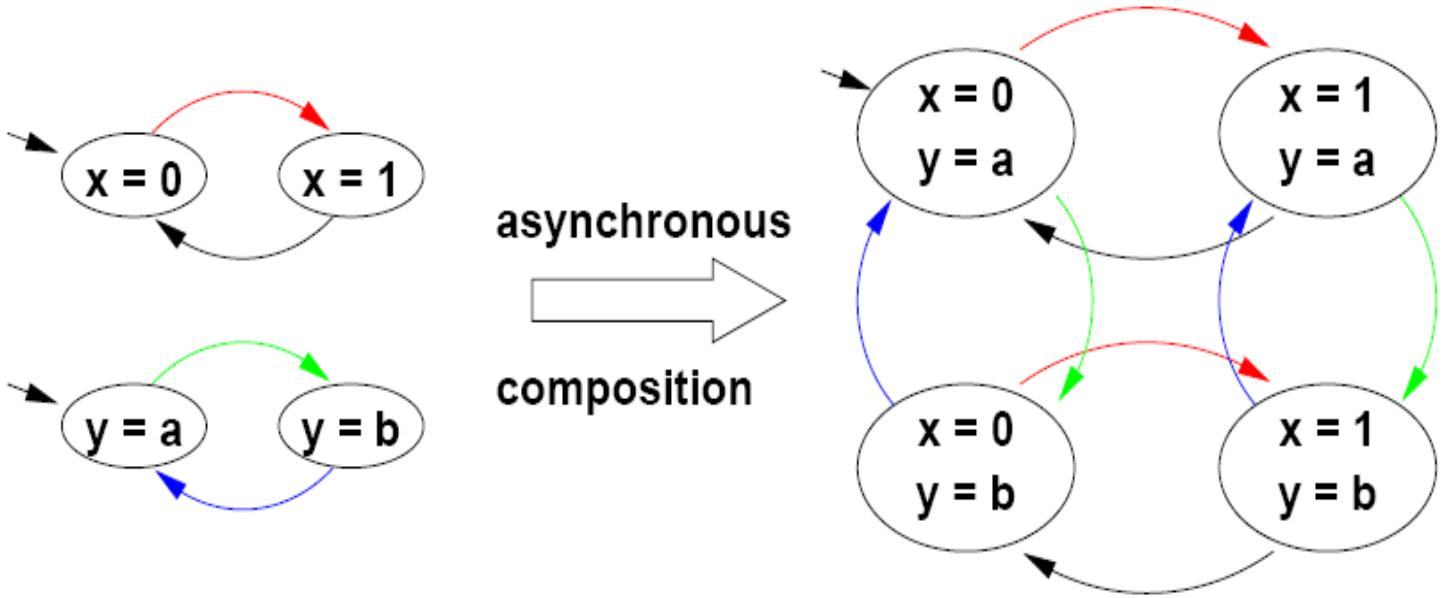
(pratiti boju strelica):



Sinkrona i asinkrona kompozicija Kripkeovih struktura

- Asinkrona kompozicija: **u svakom vremenskom trenutku jedna komponenta mijenja stanje.**
- Tipičan primjer: **komunikacijski protokol**.

(pratiti boju strelica):



Zadaci

1. Koje su od navedenih sintaksno dobro definirane formule u logici CTL:

   a) $p \wedge \neg q \vee EX\ p$

   b) $AF\ (G\ p \Rightarrow GF\ q)$

   c) $AG\ AF\ EX\ (p \wedge q)$

   d) $A((p\ U\ q) \vee (q\ U\ \neg r)$

   e) $E(p \Rightarrow E(q \Rightarrow r))$

   f) $A\ ((q \wedge r)\ U\ (p \wedge r))$

2. Preslikajte rečenice prirodnog jezika u formule logike CTL:

   a) "Uvijek u svim stanjima sustava vrijedi da p ne vrijedi dok ne počne vrijediti q."

   b) "Postoji put u kojem se konačno dolazi do stanja u kojem vrijedi p i od kojeg dalje p ne vrijedi u sljedeća dva stanja."

   c) "U početnom stanju vrijedi p, a zatim u sljedećem stanju barem na jednom putu izvođenja ne vrijedi q."

Formalna verifikacija kritičnih dijelova programa

Sustav NuSMV

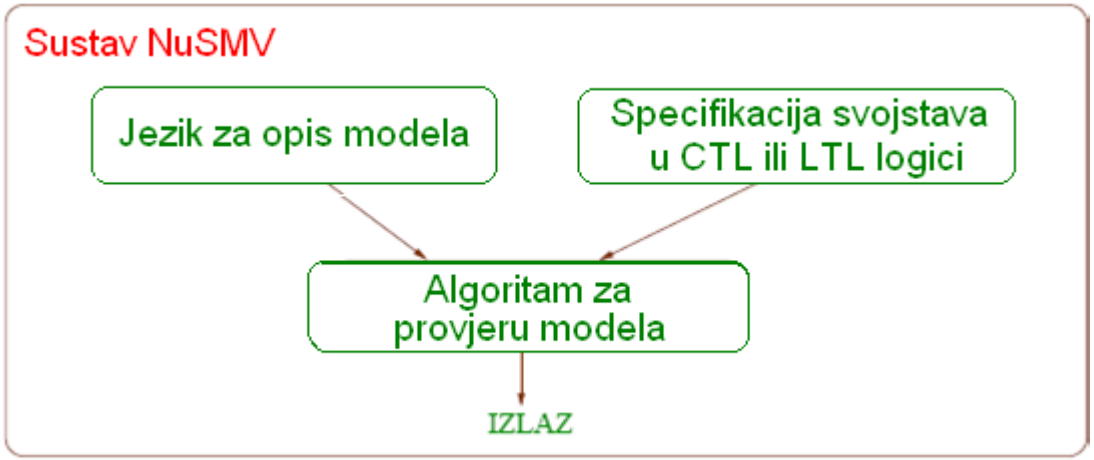
Uvod

- NuSMV
  - Sustav za simboličku provjeru modela (engl. *symbolic model checking*)
  - Nadograđena inačica izvornog sustava SMV (otuda ono Nu [nju] ☺)
  - Trenutna verzija NuSMV 2.6.0

- Izvorni SMV (engl. *Symbolic Model Verifier*)
  - Razvio ga je Ken McMillan, CMU, u doktoratu iz 1992.

Uvod

- Formalno: program koji provjerava vrijedi li:
  M |= ϕ, (implementacija M logički zadovoljava specifikaciju ϕ) pri čemu je M Kripke struktura, ϕ je svojstvo izraženo u vremenskoj logici (CTL-u ili LTL-u).
- NuSMV: verifikacija kritičnih dijelova programskih produkata



Ulaz i izlaz sustava

- **Ulaz sustava:**
  - Tekstualni opis programa u jeziku NuSMV - model sustava
  - Specifikacija svojstava u CTL ili LTL logici
    - Postoji i mogućnost specifikacije u RTCTL (Real-Time CTL)) i PSL-u - Property Specification Language)
  - U sklopu ovog kolegija na laboratorijskim vježbama koristit ćemo samo CTL logiku pri specifikaciji i provjeri svojstava u NuSMV-u

- **Izlaz sustava:**
  - '**TRUE**' - ako je specifikacija zadovoljena za sva početna stanja modela
  - '**FALSE**' - inače + ispis (engl. *trace*) koji pokazuje zašto navedeno svojstvo nije ispunjeno u modelu opisanim NuSMV programom

Algoritam za provjeru modela

- Algoritam se temelji na simboličkom prikazu stanja i prijelaza pomoću:
  - BDD-ova – binarnih dijagrama odlučivanja
    - Za logike CTL i LTL, verifikacija beskonačnih putova
  - SAT-rješavača
    - Za logiku LTL, verifikacija konačnih putova - ograničena provjera modela (engl. *Bounded model checking*)

- Teme predavanja koja slijede nakon međuispita

NuSMV: ulazni jezik

- Motivacija prilikom modeliranja
  - opis upravljanja sustavom i interakcija dijelova sustava
  - posebno služi modeliranju kritičnih programskih dijelova
  - posebno značajno za modeliranja ponašanja višedretvenih programa prilikom čega može doći do više vrsta problema (utrka, izgladnjivanje, potpuni zastoj, itd.)

- Namjena ulaznog jezika: opis relacije prijelaza (tranzicijske relacije) konačne Kripkeove strukture
  - Pomoću sustava NuSMV opisuju se međuovisni strojevi s konačnim brojem stanja (engl. *Communicating Finite State Machines*).
  - Jezik NuSMV ne podržava složene strukture podataka (liste, stabla, mape...)
- Sličan jezicima za opis sklopovlja, ali mu to nije namjena

NuSMV: programi

- Programi u NuSMV-u sastoje se od jednog ili više modula
  - Jedini posebni modul je **main** (slično kao C, Java, . . .)

- Svaki modul se obično sastoji od:
  - **Deklaracije varijabli**
  - **Dodjeljivanja vrijednosti varijablama**
  - **Svojstava koja se žele provjeriti (uglavnom u modulu main)**

Varijable (1/2)

- Varijable u NuSMV-u predstavljaju varijable stanja modela i instance pojedinih modula

- Deklaracija varijabli:

```
<decl> ::=
  "VAR"
   atom1 ":" <type> ";"
      ...
```

- Tip varijable (<type>) može biti:
  - **boolean**
  - **riječ (vektor bitova određene duljine)**
  - **enumerirani (pobrojani) tip**
  - **konačan podniz cijelih brojeva**
  - **korisnički definirani modul (instanca modula),**
  - **niz prethodno navedenih tipova**

Varijable (2/2)

- Neki primjeri deklaracija:

```
VAR
a : boolean;
w : word[7];
switch : {on, off};
state : {start, request, busy, stop};
n : 1..10000;
server6 : Server(req, stateClient);
producerWG : process Producer();
arr : array 1..20 of {start, busy, stop};
```

- Tip ***boolean*** može poprimiti vrijednosti **FALSE** ili **TRUE** (paziti: od verzije NuSMV 2.5.1 više boolean vrijednosti nisu 0 ili 1!!!)
- Ključna riječ ***process*** ispred imena modula definira asinkroni način izvođenja

Dodjeljivanja (1/4)

- Dodjeljivanje (engl. *assignment*) daje varijabli:
  - **Trenutačnu vrijednost**
  - **Inicijalnu vrijednost - *init***
  - **Sljedeću vrijednost** (na temelju trenutnih vrijednosti varijabli programa) **- *next***
- Dodjeljivanje vrijednosti varijablama:

```
<assignment> ::=
"ASSIGN"
  <dest> ":=" <expr> ";"
  ...
  <dest> ::=
    atom
  | "init" "(" atom ")"
  | "next" "(" atom ")"
```

Dodjeljivanja (2/4)

- Na lijevoj strani dodjeljivanja može se naći:
  - **atom** **označava trenutačnu vrijednost varijable,**
  - ***init(*atom)** **označava početnu vrijednost varijable,**
  - ***next*(atom) označava vrijednost varijable u sljedećem stanju**
- Izraz (<expr>) na desnoj strani dodjeljivanja može se evaluirati u:
  - **Cjelobrojnu vrijednost ili simboličku konstantu**
  - **Skup vrijednosti**

  U prvom slučaju lijeva strana jednostavno postaje desna, a u drugom lijeva strana postaje jedan od elemenata s desne strane (**nedeterminizam**)
- Primjeri dodjela:

```
ASSIGN
a := n mod 2;
init(switch) := off;
next(state) := {busy, stop};
```

Dodjeljivanja (3/4)

- Ograničenja na dodjeljivanja
  1. Svakoj varijabli dodjeljivanje može biti provedeno samo jednom i to ili u “init” i “next” bloku ili za trenutačnu vrijednost, npr:

```
init(status) := busy;
init(status) := stop;
```

nedozvoljeno

```
next(status) := busy;
next(status) := stop;
```

nedozvoljeno

```
next(status) := busy;
status := stop;
```

nedozvoljeno

```
init(status) := busy;
next(status) := stop;
```

dozvoljeno

Dodjeljivanja (4/4)

- Ograničenja na dodjeljivanja
  2. Varijabla ne može imati cikluse u svojem grafu ovisnosti koji nisu odvojeni s kašnjenjem, npr:
     - `x := (x + 1) mod 2;` nedozvoljeno
     - `next(x) := x & next(x);` nedozvoljeno
     - `next(x) := x & next(y);` nedozvoljeno
     - `next(y) := y & next(x);`
     - `next(x) := x & next(y);` dozvoljeno
     - `next(y) := y & x;`

Specifikacija svojstava (1/2)

- Specifikacija ponašanja sustava zadana je formulom u vremenskoj logici CTL
- Zapis CTL-formula u sustavu NuSMV :

```
<form> ::= "CTLSPEC" <ctlform> -- osim CTLSPEC može i SPEC

<ctlform> ::=
<expr> -- Booleova formula
| "!" <ctlform> -- logička negacija
| <ctlform1> "&" <ctlform2> -- logički i
| <ctlform1> "|" <ctlform2> -- logički ili
| <ctlform1> "->" <ctlform2> -- logička implikacija
| <ctlform1> "<->" <ctlform2> -- logička ekvivalencija
| "E" <pathform> -- egzistencijalni kvantifikator puta
| "A" <pathform> -- univerzalni kvantifikator puta
```

Specifikacija svojstava (2/2)

- Zapis CTL-formula u sustavu NuSMV (nastavak):

```
<pathform> ::=
   "X" <ctlform> -- operator sljedećeg stanja
 | "F" <ctlform> -- operator konačnog dolaska u stanje
 | "G" <ctlform> -- operator cijelog puta
 | "[" <ctlform1> "U" <ctlform2> "]" -- operator kod kojeg
   na cijelom putu ctlform1 vrijedi dok ne počne ctlform2
   vrijediti
```

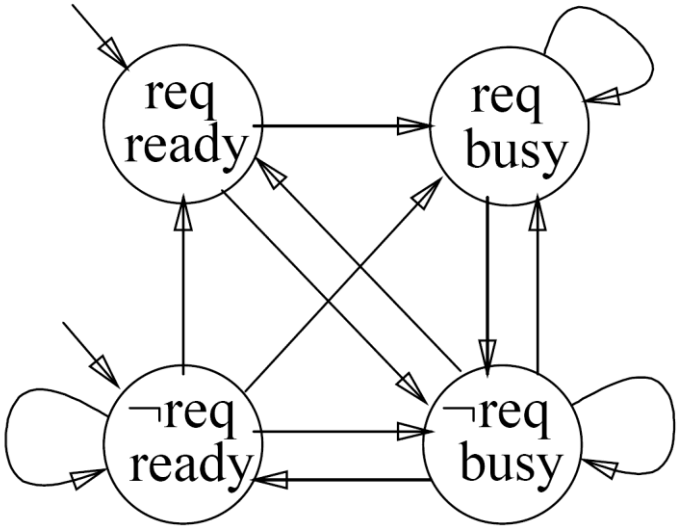
- Svaka CTL-formula definira se novom ključnom riječi CTLSPEC

- Primjeri specifikacija:

```
CTLSPEC AG AF state=start
CTLSPEC EF n>=40
CTLSPEC E [ b=TRUE U state=busy ]
CTLSPEC AG (state=request -> AF state=busy)
```

NuSMV: primjer glavnog programa

```
MODULE main
VAR
  request : boolean;
  status : {ready,busy};
ASSIGN
  init(status) := ready;
  next(status) := case
        request : busy;
        TRUE : {ready,busy};
  esac;
CTLSPEC
   AG (request -> AF status =
   busy)
```



FSM se sastoji od četiri stanja - svako stanje odgovara mogućoj vrijednosti dviju binarnih varijabli

Varijable:

request - varijable tipa boolean, TRUE ili FALSE

status - je enumerirana varijabla koja može poprimiti vrijednost ready ili busy

Inicijalna i *sljedeće* vrijednosti varijable ***request*** nisu određene u programu. Na taj je način modeliran utjecaj okoline (NuSMV sustav može pridijeliti proizvoljne vrijednosti toj varijabli).

Varijabla ***status*** je parcijalno određena: inicijalno poprima vrijednost *ready*, i postaje *busy* ako je ***request*** jednak *TRUE*. Inače ako je request jednak *FALSE* vrijednost varijable nije određena.

Nedeterminističke vrijednosti

- Varijabla može nedeterministički poprimiti vrijednost na dva osnovna načina:
  - **Implicitno – varijabli se ne dodjeljuje nikakva vrijednost (ulazna varijabla)**
  - **Eksplicitno – nedeterminističkim dodjelama: varijabli se dodjeljuje skup vrijednosti (varijabla nedeterministički poprima jednu vrijednost iz skupa)**
- Prethodni primjer implicitnog nedeterminizma
  - **Utjecaj varijable *request* primjer je implicitnog nedeterminizma**
- Primjer eksplicitnog nedeterminizma :
  - ***next*(status) := {ready, busy};**
- Nedeterminističke dodjele – koriste se za modeliranje utjecaja okoline, nepotpune implementacije ili za formiranje apstrakcija složenih protokola, gdje se ne može utvrditi točna vrijednost varijable stanja

Moduli u NuSMV-u

- NuSMV podržava opis sustava razbijanjem na više modula (omogućuje provjeru svojstava interakcije između dijelova sustava)
- Modul se instancira tako da se deklarira varijabla čiji je tip ime modula
- Pristup varijablama unutar modula obavlja se pomoću točke "."
  - **m.v - *pristupa se varijabli v unutar modula m***

Primjer: Trobitni brojač

```
MODULE main
VAR
   bit0 : counter_cell(TRUE);
   bit1 : counter_cell(bit0.carry_out);
   bit2 : counter_cell(bit1.carry_out);
CTLSPEC
   AG AF bit2.carry_out

MODULE counter_cell(carry_in)
VAR
   value : boolean;
ASSIGN
   init(value) := FALSE;
   next(value) := value xor carry_in;
DEFINE
   carry_out := value & carry_in;
```

NuSMV primjer: Trobitni brojač

- Ovaj program opisuje brojač koji uzastopno broji od 000 do 111
- Trobitni brojač je opisan pomoću tri jednobitna brojača:
  - **Modul `counter_cell` je instanciran tri puta: `bit0`, `bit1` i `bit2`.**
  - **Svaki modul ima jedan formalni parametar: `carry_in`**
  - **Moduli su međusobno povezani tako da je `carry_in` modula `bit1` ujedno `carry_out` modula `bit0` (analogno vrijedi za `bit2` i `bit1`)**

- Ključna riječ **DEFINE** se koristi kao makro za dodjelu izraza **`value & carry_in`** simbolu **`carry_out`**
- Značenje ključne riječi **DEFINE** je to da gdje god da se dalje u programu pojavi **`carry_out`**, bit će zamijenjen s izrazom **`value & carry_in`**
- Ključna riječ **DEFINE** ne proširuje prostor stanja
- DEFINE ne može raditi s nedeterminističkim vrijednostima

Sinkrono i asinkrono izvršavanje u NuSMV-u

- Sinkroni način izvršavanja - postoji globalni sat, i svakim taktom globalnog sata, svaki od modula (ASSIGN blok) se izvršava paralelno

- Asinkroni način izvršavanja - pojedini dijelovi sustava (moduli) odvijaju se različitim brzinama, pri čemu se njihova izvođenja proizvoljno isprepliću (engl. *interleaving execution*)
  - **U svakom taktu, proizvoljno se odabire jedan od modula i izvršava se – unutarnja varijabla *running* za taj modul se postavlja na TRUE)**
  - **Ukoliko se želi da se neki modul izvršava u *interleaved* načinu izvođenja potrebno je postaviti ključnu riječ *process* prilikom instanciranja modula**
  - Korisno za opis komunikacijskih protokola, asinkronih sklopova i ostalih sustava koji nisu sinkronizirani pomoću nekog globalnog sata te za modeliranje višedretvenosti

Međusobno isključivanje (engl. *mutual exclusion*, MUTEX)

- Ako konkurentni procesi (procesi koji se istovremeno izvode) dijele određeni resurs (datoteka na disku, zapis u bazi podataka, podatak u zajedničkoj memoriji), nužno je osigurati da procesi ne mogu istovremeno pristupati dijeljenom resursu
- Potrebno je definirati *kritične odsječke* unutar svakog od procesa i osigurati da samo jedan proces smije istovremeno biti u kritičnom odsječku
- Problem: pronalaženje protokola kojim se utvrđuje kad koji od procesa smije ući u svoj kritični odsječak

Svojstva protokola MUTEX

Sigurnost (engl. *safety*)

- **Protokol osigurava da samo jedan proces bude u kritičnom odsječku u bilo kojem trenutku**

Životnost (engl. *liveness*)

- **Kada god neki od procesa želi ući u kritični odsječak, uvijek će konačno i ući**

Svojstvo neblokiranja (engl. *non-blocking*)

- **Proces uvijek može zahtjevati ulazak u kritični odsječak, drugi ga proces ne smije u tome sprečavati**

Svojstvo nedeterminiranog redoslijeda (engl. *no strict sequencing*)

- **Procesi mogu ulaziti u svoje kritične odsječke na proizvoljan način (ne postoji predodređeni redoslijed ulaska u kritični odsječak)**

Primjer protokola MUTEX u NuSMV-u (1/3)

```
MODULE main
VAR
   pr1: process proc(pr2.st, turn, FALSE);
   pr2: process proc(pr1.st, turn, TRUE);
   turn: boolean;
ASSIGN
   init(turn) := FALSE;
   ...
```

- Varijabla **turn** određuje koji proces smije ući u kritični odsječak
- Dvije instance modula **proc**, koji opisuje kontrolne aspekte konkurentnih procesa

Primjer protokola MUTEX u NuSMV-u (2/3)

```
...
MODULE proc(other-st, turn, myturn)
VAR
   st: {n, t, c};
ASSIGN
   init(st) := n;
   next(st) :=
   case
       (st = n) : {t,n};
       (st = t) & (other-st = n) : c;
       (st = t) & (other-st = t) & (turn = myturn): c;
       (st = c) : {c,n};
      TRUE : st;
   esac;
...
```

st – interno stanje procesa

[n]on-critical – proces nije u kritičnom odsječku

[t]rying – proces želi ući u kritični odsječak

[c]ritical – proces je u kritičnom odsječku

Ako je proces u nekritičnom odječku, može u njemu ostati ili preći u stanje **t** (trying – želi ući u kritični odsječak)

Ako je proces u stanju **t**, a drugi proces je u nekritičnom odsječku, tada proces može ući u kritični odsječak (preći u **c**)

Ako je proces u stanju **t**, i drugi proces je također u tom stanju, tada proces može ući u kritični odsječak (preći u stanje **c**) samo ako je njegov red (turn=myturn)

Ako je proces u stanju **c** (kritični odsječak) onda može ostati u tom stanju ili izaći iz kritičnog odsječka (preći u stanje **n**).

Primjer protokola MUTEX u NuSMV-u (3/3)

```
...
next(turn) :=
    case
        turn = myturn & st = c : !turn;
        TRUE : turn;
    esac;
JUSTICE running
JUSTICE !(st = c)
```

- Ako je proces u kritičnom odsječku i varijabla **turn** je pokazivala da je njegov red za izvođenje tada se negira njena vrijednost i prebacuje pravo na drugi proces.
- Varijabla **turn** određuje koji će proces ući u kritični odsječak ako u jednom trenutku oba procesa žele ući u kritični odsječak (**st=t**).

Pravednost (engl. *fairness*) u NuSMV-u

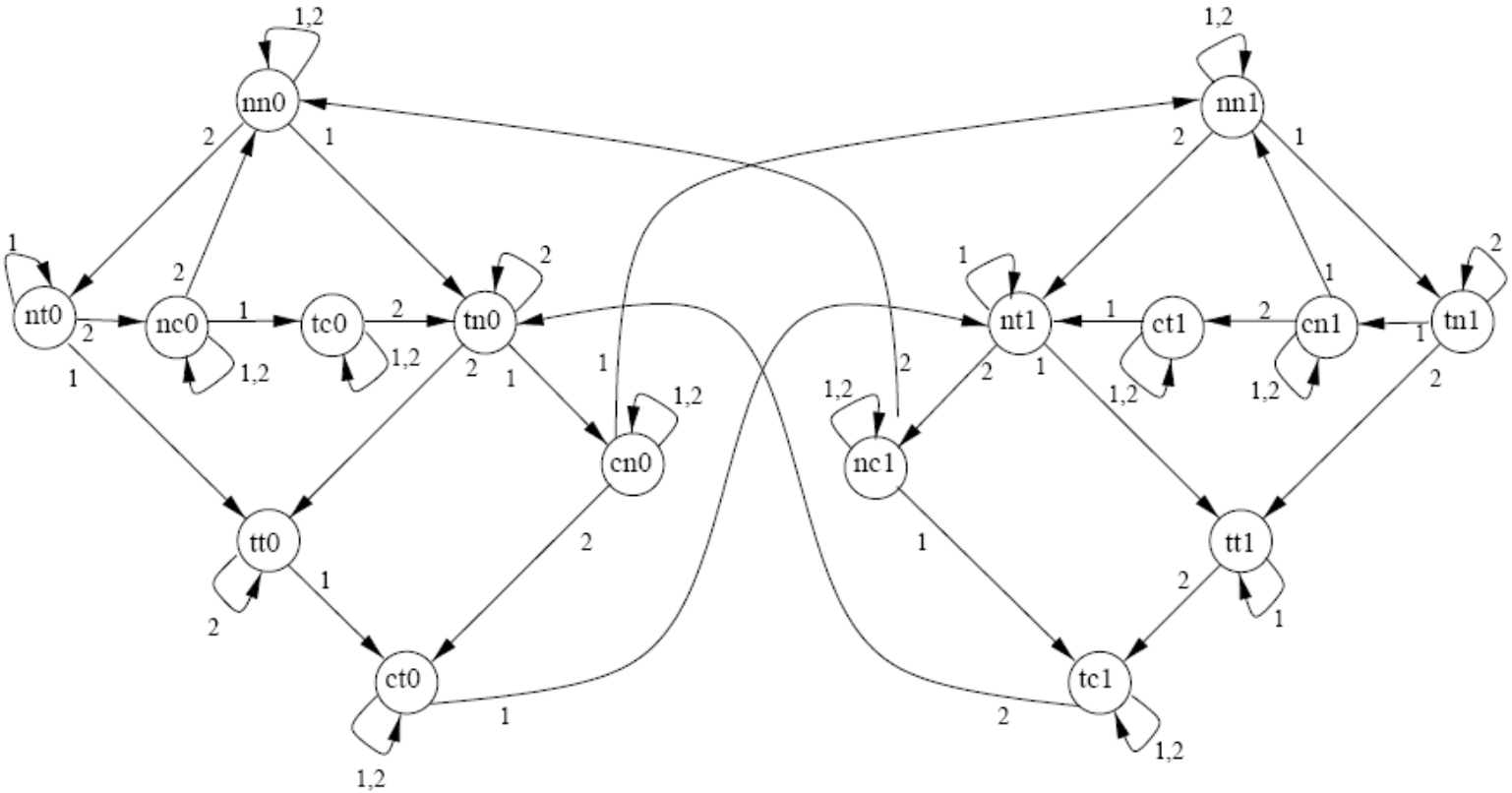
- Važno svojstvo sustava NuSMV:
  - **moguće je ograničiti područje pretraživanja prostora stanja na samo one staze izvršavanja duž kojih je proizvoljna CTL formula istinita beskonačno često**
- Obično te staze izvršavanja modeliraju ***pravedan*** *(*engl. *fair)* pristup određenom resursu (preciznije, radi se o Büchiovom tipu pravednosti).
- Ograničenje pravednosti sastoji se od:
  - **Ključne riječi `JUSTICE` (može i `FAIRNESS`)**
  - **CTL formule *f***
- Prilikom provjere nekog svojstva, NuSMV će ignorirati svaku stazu na kojoj formula ***f*** ne vrijedi beskonačno često.
- **`JUSTICE running`** – označava da se svaki proces treba izvršavati beskonačno često

Pravednost u primjeru MUTEX

- **`JUSTICE running`**
  - Ako je modul koji se razmatra deklariran pomoću ključne riječi *process*, tada će u svakom trenutku NuSMV nedeterministički odrediti hoće li se taj modul izvršavati ili ne. U ovom slučaju razmatrat će se samo ona izvršavanja za koja vrijedi da će modul biti beskonačno često izabran za izvođenje.

- **`JUSTICE !(st = c)`**
  - Razmatraju se samo oni putevi na kojima modul nije beskonačno dugo u kritičnom odsječku. Naime, u modelu je (zbog nedeterminizma) moguće da pojedini modul ostane u svom kritičnom odsječku koliko god želi. To je trivijalan slučaj zbog kojeg može doći do kršenja svojstva životnosti.

FSM za primjer MUTEX-a



Jezik NuSMV korišten u stilu ograničenja (engl. *constraint*)

- U NuSMV-u moguće je osim stila dodjeljivanja (engl. *assignment*) koristiti i **stil ograničenja**:

  INIT <simp_expr> umjesto ***init(atom) := <simp_expr>;***
  TRANS <next_expr> umjesto ***next(atom) := <simp_expr>;***
  INVAR <simp_expr> umjesto ***atom := <simp_expr>;***

- Primjeri prevođenja stila dodjeljivanja u stil ograničenja:

```
init(state):= {ready,busy};  INIT state in {ready,busy}
next(state):= ready;         TRANS next(state) = ready
out := b2 + 3*b1;            INVAR out = b2 + 3*b1
```

- Ograničenjima zapravo navodimo logičke (*boolean*) tvrdnje koje trebaju vrijediti u modelu
- Svaki modul može imati nula, jedno ili više ograničenja, a moguće je i kombinirati dodjeljivanja i ograničenja

Napomene oko korištenja stila ograničenja

- Rizik definiranja nekonzistentnih početnih ograničenja, npr.
  `INIT (p & !p)` -- nepostojanje inicijalnog stanja - bilo koja specifikacija će biti zadovoljena
- Rizik definiranja nekonzistentnih ograničenja na prijelaze
  - Relacija prijelaza Kripkeove strukture nije potpuna (iz nekog stanja nisu specificirani daljnji prijelazi), npr.

```
MODULE main
VAR b : boolean;
TRANS b = TRUE -> FALSE   -- umjesto: ... -> next(b) = FALSE
```

- Nedeterminizam je često skriven u ograničenjima, npr.

```
next(state) := case
(state = ready & request = TRUE) : busy;
TRUE : { ready, busy }; esac
=>
TRANS (state = ready & request = TRUE) -> next(state) =
busy
```

Primjer: najveći zajednički djelitelj (1/3)

- Neka je zadan program u C-u:

```
void main() {
... // inicijalizacija brojeva a i b, pretpostavka da su oba pozitivna
    while (a!=b) {          // oznaka L1
        if (a>b)            // oznaka L2
            a=a-b;      // oznaka L3
        else
            b=b-a;      // oznaka L4
    }
    ... // NZD=a=b         // oznaka L5
}
```

Primjer: najveći zajednički djelitelj (2/3)

- Program preveden u NuSMV, u stil dodjeljivanja:

```
MODULE main()
VAR a: 0..100; b: 0..100;
pc: {L1,L2,L3,L4,L5};
ASSIGN
  init(pc):=L1;
  next(pc):= case
      pc=L1 & a!=b: L2;
      pc=L1 & a=b: L5;
      pc=L2 & a>b: L3;
      pc=L2 & a<=b: L4;
      pc=L3 | pc=L4: L1;
      pc=L5: L5;
  esac
...
```

```
...
next(a):= case
  pc=L3: a-b;
  TRUE: a;
  esac
next(b):= case
  pc=L4: b-a;
  TRUE: b;
  esac
```

Primjer: najveći zajednički djelitelj (3/3)

- U NuSMV-u, stil ograničenja (primijetiti da više sliči originalnom programu)

```
MODULE main
VAR a : 0..100; b : 0..100; pc : {L1, L2, L3, L4, L5};
INIT pc = L1
TRANS
pc = L1-> (((a != b & next(pc) = L2) | (a = b & next(pc) = L5))
                        & next(a) = a & next(b) = b)
TRANS
pc = L2-> (((a > b & next(pc) = L3) | (a < b & next(pc) = L4))
                        & next(a) = a & next(b) = b)
TRANS
pc = L3 -> (next(pc) = L1 & next(a) = (a - b) & next(b) = b)
TRANS
pc = L4 -> (next(pc) = L1 & next(b) = (b - a) & next(a) = a)
TRANS
pc = L5 -> (next(pc) = L5 & next(a) = a & next(b) = b)
```

Pokretanje sustava NuSMV

- NuSMV je moguće pokrenuti na dva načina: uobičajen (*default*) i interaktivan
- Uobičajen način:
    - Provjera svojstava svih CTL specifikacija zadanih u ulaznoj datoteci odjednom, nema mogućnost podešavanja
    - Sintaksa poziva: > NuSMV <put_do_datoteke.smv>
- Interaktivan način:
    - Omogućuje podešavanja parametara algoritma verifikacije, provjeru konačnog automata, simulaciju prolazaka kroz stanja procesa, provjeravanje jedne po jedne specifikacije...
    - Sintaksa poziva: > NuSMV –int
    - Učitavanje modela: > read_model –i <put_do_datoteke.smv>

- U okviru 1. domaće zadaće studenti se upoznaju s oba načina rada, ali samo s jednostavnim funkcionalnostima interaktivnog načina

1. zadatak

Za zadani model u NuSMV-u nacrtati Kripkeovu strukturu, te odrediti istinitost sljedećih CTL izraza:

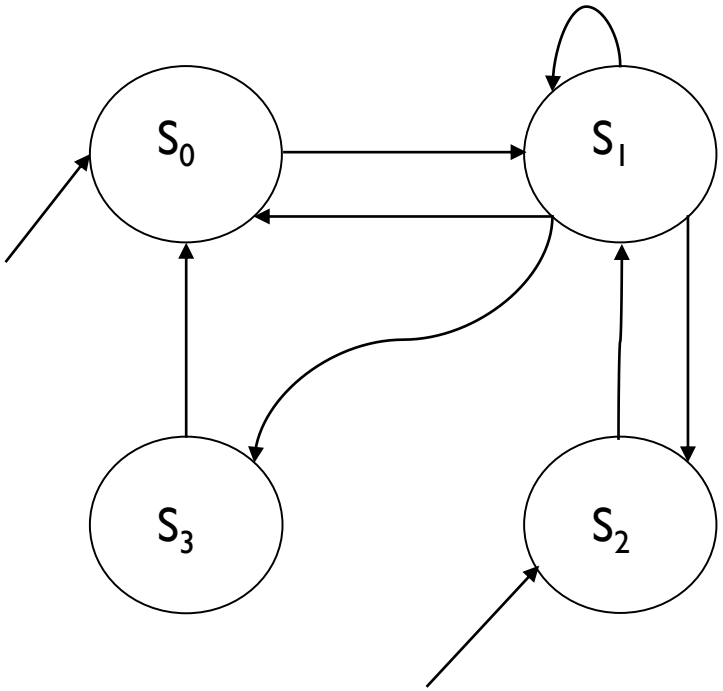
a) AG(AF(a & b)) b) EX(!a & !b)

```
MODULE mod
VAR a : boolean;
    b : boolean;
ASSIGN
    init(a) := FALSE;
    init(b) := FALSE;
    next(a) :=
            case
                (b = FALSE) :  a;
                TRUE : !a;
            esac;
    next(b) :=
            case
                TRUE : !b;
            esac;
```

2. zadatak

Za zadanu Kripkeovu strukturu napisati kompletan NuSMV-kôd.

Pomoću ograničenja pravednosti osigurati da se sustav beskonačno često nađe u stanju S_2



S_0 = {req=FALSE, stat=FALSE}

S_1 = {req=FALSE, stat=TRUE}

S_2 = {req=TRUE, stat=FALSE}

S_3 = {req=TRUE, stat=TRUE}

3. zadatak

Za zadani kôd modula u NuSMV-u potrebno je nacrtati odgovarajuću Kripkeovu strukturu i odrediti istinitost CTL specifikacija.

```
MODULE igrac_3
VAR
          ready : boolean;
          turn : {one, two, three};
ASSIGN
          init (ready) := TRUE;
          init (turn) := one;
          next (ready) :=
                    case
                              (!ready & (turn = one | turn = two)) : TRUE;
                              (!ready & turn = three) : FALSE;
                              TRUE : {FALSE,TRUE};  esac;
          next (turn) :=
                    case
                              (!ready & turn = one) : two;
                              (!ready & turn = two) : {one,two,three};
                              (ready & turn = one) : {one,two};
                              TRUE: three;  esac;

A)  CTLSPEC AF AG (turn = three)
B)  CTLSPEC EF AX ready
```

Oko 1. domaće zadaće

- **1. domaća zadaća uključuje**
  - Ispitivanje svojstava MUTEX-a na par različitih implementacija korištenjem specifikacija pisanih u CTL-u
  - Ispitivanje pravednosti istih implementacija
  - Provjera modela nekoliko različitih zadataka
  - Interaktivno pokretanje NuSMV-a i provjeru modela na nekoliko primjera
  - Nekoliko zadataka vezanih uz sustav NuSMV
- **Upute i rokovi za 1. domaću zadaću dostupni su na web stranicama predmeta**

FVPP: Vremenska logika i računanje skupova stanja

Linearna vremenska logika (LTL)

Potpuna logika vremenskog grananja (CTL*)

Eksplicitni postupci računanja skupova stanja koja zadovoljavaju CTL specifikaciju

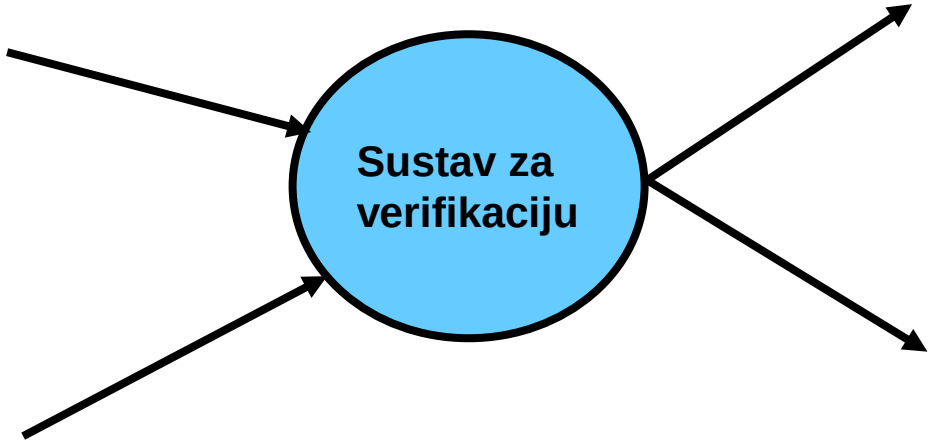
Pripremio: izv. prof. dr. sc. Alan Jović

Ak. god. 2022./2023.

Provjera modela (engl. *model checking*)

I **= Implementacija (model sustava koji se verificira). Izraženo povezanim strojevima s konačnim brojem stanja (FSM).**

DA = model sustava logički zadovoljava specifikaciju



NE (+ ispis traga (engl. *trace*) pogrešnog izvođenja programa)

S **= Specifikacija (željeno ponašanje). Izraženo najčešće u vremenskoj logici - CTL ili LTL**

Simbolički opisujemo:

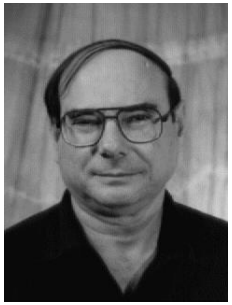
$$\boxed{I \models S}$$

FVPP: Propozicijska linearna vremenska logika

Engl. *Propositional Linear Temporal Logic*

PLTL (najčešće samo LTL)

A. Pnueli: "The temporal logic of programs" (1977.)



Pretpostavka primjene (vrijede i za CTL)

- Promatramo sustave koji se mogu modelirati strojevima s **konačnim brojem stanja**
- Promatramo **reaktivne programe (neterminirajuće)** – kontinuirano reagiraju na okolinu (operacijski sustavi, sustavi upravljanja procesima i sl.)
- Analiza ponašanja duž **potencijalno beskonačnih putova izvođenja**.

Prisjetimo se...

- Kontekst vremenske logike CTL:
- Kripkeova struktura (model M) promatra se kao beskonačno stablo počevši od početnog stanja s0 (“stablo se odmota”).
- U svakom stanju moguć je prijelaz u jedno od **mogućih više stanja**.
- Eksplicitno se kvantificiraju svi putovi izvođenja (**A**) ili barem jedan put izvođenja (**E**).
- Formula vremenske logike CTL je istinita ako je istinita barem za jedan put izvođenja (**E**) ili za sve putove izvođenja (**A**), ovisno o vrsti specifikacije.

Kontekst vremenske logike LTL

- Kripkeova struktura (M) se dekomponira u **pojedinačne beskonačne sekvence** – putove izvođenja programa.

- Ponašanje sustava je **kolekcija beskonačnih sekvenci prijelaza** (engl. *infinite transition sequences*)

- Svako stanje u pojedinoj sekvenci ima **samo jednog sljedbenika**.

sekvenca 1: $s0 \rightarrow s1 \rightarrow s2 \rightarrow s3 \rightarrow s4 \rightarrow s5 \rightarrow s6 \rightarrow s7 \rightarrow \ldots$

sekvenca 2: $s0 \rightarrow s1 \rightarrow s2 \rightarrow s7 \rightarrow s4 \rightarrow s5 \rightarrow s6 \rightarrow s2 \rightarrow \ldots$

sekvenca *n*: . . .

- LTL formula je istinita za neki sustav (Kripkeovu strukturu) ako vrijedi za sve pojedinačne putove izvođenja (za sve sekvence) toga sustava. Kvantifikator "A" je implicitan. Nema kvantifikatora "E".

Linearna vremenska struktura

- Jedna beskonačna sekvenca s početnim stanjem i označavanjem propozicijskih simbola koji vrijede u pojedinim stanjima naziva se **linearna vremenska struktura**.

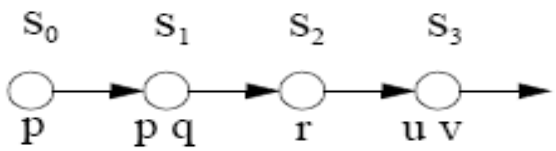
- Formule logike LTL se interpretiraju po svim beskonačnim sekvencama (linearnim vremenskim strukturama).

- AP: skup atomičkih propozicijskih simbola
- **Linearna vremenska struktura** dana je trojkom $\pi = (S, x, L)$

S:	konačan skup stanja
$x: N \rightarrow S$	beskonačna sekv. prijelaza = vremenska crta ($x = s_0, s_1, \ldots$)
$L: S \rightarrow 2^{AP}$	označavanje stanja skupom propozicijskih simbola.

Primjer:

x:

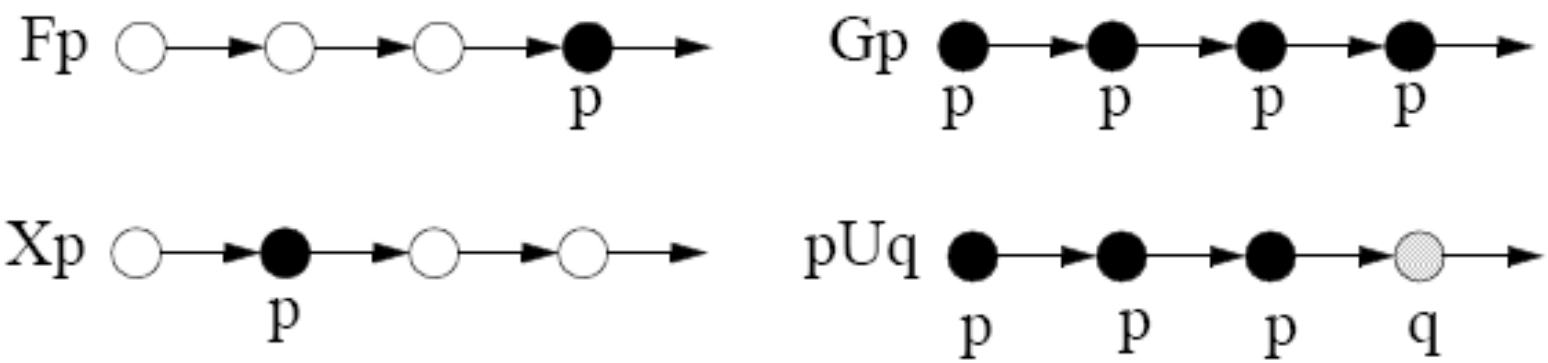


$L(s_0) = \{p\}, L(s_1) = \{p, q\}, L(s_2) = \{r\}, L(s_3) = \{u, v\}, \ldots\ldots$

Vremenski operatori u LTL-u

Uporaba vremenskih operatora (bez kvantifikatora):

F*p* ("eventually *p*", "finally *p*")	- konačno p
G*p* ("always *p*", "henceforth *p*")	- uvijek p
X*p* ("next time *p*")	- u sljedećem koraku p
*p***U***q* ("*p* until *q*")	- p dok ne počne vrijediti q



Provjera modela: implicitno provjeravamo **sve** putove u Kripkeovoj strukturi (kvantifikator A je implicitan).

Formalna sintaksa (P)LTL-a (1/2)

(1) Atomičke propozicije su formule.
(2) Ako su p i q formule, tada su $p \wedge q$, $\neg p$, p U q, Xp formule.

Ostale formule mogu se izvesti:

$p \vee q$	ekvivalentan oblik	$\neg(\neg p \wedge \neg q)$,
$p \Rightarrow q$	ekvivalentan oblik	$\neg p \vee q$,
$p \equiv q$	ekvivalentan oblik	$(p \Rightarrow q) \wedge (q \Rightarrow p)$,
true	ekvivalentan oblik	$p \vee \neg p$,
false	ekvivalentan oblik	$\neg$*true*,
Fp	ekvivalentan oblik	(*true* U p),
Gp	ekvivalentan oblik	$\neg$F$\neg p$.

LTL dozvoljava Booleove kombinacije i ugniježđivanje vremenskih operatora

Formalna sintaksa LTL-a (2/2)

Prioritet unarnih i binarnih operatora:

1. Unarni operatori ($\neg$, X, F, G) povezuju najčvršće
2. U - binarni operator
3. $\wedge$ - konjunkcija
4. $\vee$ - disjunkcija
5. $\Rightarrow$ - implikacija

Primjer suvišnih zagrada:

$(F\ (p \Rightarrow\ (G\ r))\ \vee\ ((\neg q)\ U\ p))\ \equiv\ F\ (p \Rightarrow\ G\ r)\ \vee\ \neg q\ U\ p$

Preporuka: **upotrebljavaj zagrade zbog jasnijeg razumijevanja**, iako mogu biti suvišne.

Formalna semantika (P)LTL-a

Formula φ dana u vremenskoj logici LTL ima značenje u odnosu na **svaku pojedinu linearnu vremensku strukturu** $\pi = (S, x, L)$.

$\pi, x \models \varphi$ - u strukturi π formula φ je istinita za vremensku crtu **x**.
Crta (put) **x** <u>logički zadovoljava</u> ($\models$) formulu φ.

Neka je:

x - put (vremenska crta) koja započinje u s_0 , $x = s_0, s_1, \ldots$

x_i - put (vremenska crta) koja započinje u s_i , $x_i = s_i, s_{i+1}, \ldots$

Tada:

$\pi, x \models a$	akko	$a \in L(s_0)$, AP **a je istinit u** s_0.
$\pi, x_i \models b$	akko	$b \in L(s_i)$, AP **b je istinit u** s_i .
$\pi, x \models \varphi \wedge \psi$	akko	$\pi, x \models \varphi$ i $\pi, x \models \psi$
$\pi, x \models \neg\varphi$	akko	put **x** u strukturi π ne zadovoljava φ
$\pi, x \models \varphi \mathbf{U} \psi$	akko	$\exists_i\ (x_i \models \psi)$ i $\forall_{k<i}\ (x_k \models \varphi)$
$\pi, x \models \mathbf{X}\varphi$	akko	$x_1 \models \varphi$ u sljedećem stanju je $\varphi = \mathbf{T}$
$\pi, x \models \mathbf{F}\varphi$	akko	$\exists_i\ (x_i \models \varphi)$ postoji neko stanje gdje je $\varphi = \mathbf{T}$
$\pi, x \models \mathbf{G}\varphi$	akko	$\forall_i\ (x_i \models \varphi)$ u svakom stanju $\varphi = \mathbf{T}$

Konvencija: **sadašnje stanje je uključeno u buduće stanje.**

Načini iskazivanja beskonačnosti u LTL-u (engl. *LTL infinitary modalities*)

p: formula u LTL-u

F^{∞} *p* ≡ GF *p* ("globally finally *p*", "infinitely often *p*")

$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow s_7 \rightarrow \ldots$

$\neg p \quad \neg p \quad p \quad \neg p \quad p \quad \neg p \quad p \quad \neg p$

"infinitely often *p*" – "**beskonačno često se pojavljuje p**"

G^{∞} *p* ≡ FG *p* ("finally globally *p*", "almost everywhere *p*")

$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow s_7 \rightarrow \ldots$

$\neg p \quad \neg p \quad \neg p \quad \neg p \quad p \quad p \quad p \quad p$

finite ¬p **infinite p**

"Konačno globalno se pojavljuje p": nakon konačnog broja stanja u kojime p ne mora biti istinit, slijedi beskonačan niz stanja u kojima je p=TRUE.

Ispravnost LTL-formula

Primjeri sintaktički ispravnih LTL-formula:

$p \Rightarrow Fq$

$G\ (p \Rightarrow Fq)$

$p\ U\ (q\ U\ r)$

$XXG\ p$

$[\ p \wedge G\ (p \Rightarrow Xp)\ ]\ \Rightarrow\ G\ p$

Primjeri sintaktički neispravnih LTL-formula:

$U\ r$ (U nije unarni operator)

$p\ G\ q$ (G nije binarni operator)

$EG\ p$ (E kao kvantifikator ne postoji u logici LTL)

$A\ p\ U\ q$ (A se ne navodi eksplicitno kao kvantifikator u logici LTL)

Značajne valjanosti u LTL-u

Vrijede za sve linearne vremenske strukture $\pi = (\mathbf{S}, \mathbf{x}, \mathbf{L})$

Neka su p, q formule u LTL-u.

$\models G \neg p \equiv \neg F p$ $\qquad$ $\models F \neg p \equiv \neg G p$

$\models X \neg p \equiv \neg X p$

$\models F^{\infty} \neg p \equiv \neg G^{\infty} p$ $\qquad$ $\models G^{\infty} \neg p \equiv \neg F^{\infty} p$

GF = $F^{\infty} p$ – "beskonačno često p

FG = G^{∞} – "konačno (nakon nekog vremena) globalno (stalno) p"

$\models p \Rightarrow F p$ $\qquad$ $\models G p \Rightarrow p$

$\models X p \Rightarrow F p$ $\qquad$ $\models G p \Rightarrow X p$

$\models G p \Rightarrow F p$ $\qquad$ $\models G p \Rightarrow XG p$

$\models p \, U \, q \Rightarrow F q$ $\qquad$ $\models G^{\infty} q \Rightarrow F^{\infty} q$

(zadnja izjava: ako nakon nekog stanja globalno q, tada i beskonačno često q)

Distributivnost i rekurzije u LTL-u

Distribucija preko logičkih vezica:

$\models \; G\,(p \wedge q) \quad \equiv \quad (G\,p \;\wedge\; G\,q)$

$\models \; (p \wedge q)\,U\,r \quad \equiv \quad ((p\,U\,r) \wedge (q\,U\,r))$

analogno za X, F

Primijetiti da u CTL-u takva distribucija nije dozvoljena:

CTL: $A[(p \wedge q)\,U\,r\,] \neq A[(p\,U\,r) \wedge (q\,U\,r)\,]$

Rekurzivni izrazi:

$\models \; F\,p \quad \equiv \quad p \;\vee\; XF\,p$

$\models \; G\,p \quad \equiv \quad p \wedge XG\,p$

$\models \; (p\,U\,q) \;\equiv\; q \vee (p \wedge X\,(p\,U\,q))$

Preslikavanje rečenica prirodnog jezika u LTL (1/2)

1. Nije moguće doći u stanje gdje je *start* istinito a *spreman* nije istinito.

   $G\neg(start \wedge \neg spreman)$

2. Za svako stanje vrijedi: ako se pojavi zahtjev, on će konačno biti *prihvaćen*.

   $G(pojavio_zahtjev \Rightarrow F\ prihvaćen_zahtjev)$

3. Neki proces je omogućen beskonačno često na svim putovima izvođenja.

   $GF\ proces_omogućen$

4. U svakom slučaju, određeni proces će permanentno biti zaustavljen.

   $FG\ proces_zaustavljen$

5. Iz svakog stanja sustava **moguće** je doći u stanje gdje vrijedi *reset*.

   To se ne može izreći LTL-logikom, jer LTL ne može izravno potvrditi postojanje (moguće) nekog određenog puta ili putova (to naravno može CTL kvantifikatorima puta A i E).

Preslikavanje rečenica prirodnog jezika u LTL (2/2)

6. Lift **može** ostati stajati na …

Ove sve izjave ne mogu se izreći LTL-logikom, jer LTL ne može izravno potvrditi postojanje (moguće, može) nekog određenog puta ili putova (to naravno može CTL kvantifikatorima puta A i E).

Problem se djelomično može riješiti **negacijom upita**:

- Provjera postoji li put iz *s* koji zadovoljava LTL-formulu φ svodi se na provjeru da li svi putovi zadovoljavaju $\neg\varphi$.
- Ako svi putovi zadovoljavaju $\neg\varphi$, onda ne postoji put koji zadovoljava φ.
- Međutim, provjera značajki sustava u kojem postoji mješavina egzistencijskih i univerzalnih kvantifikatora ne može se riješiti na gornji način (negacija opet daje mješavinu E i A).

Proširenja LTL-a

1. **LTL s konačnim linearnim vremenskim strukturama** (odmotavamo samo dio puta: prvih *k* stanja – koristi se za ograničenu provjeru modela (engl. *bounded model checking*).
2. Promjena semantike modaliteta:

   2.1. (p U q) je istinita sve dok vrijedi ($p \wedge \neg q$), tj.

   q može stalno biti neistinit, tj. ne mora biti istinit negdje u budućnosti ("weak until"): (p W q)

   2.2. (p U q) je istinita akko u budućem trenutku (**ne sada**)

   q je istinit (q mora biti istinit **striktno u budućnosti**).
3. Proširenje s logikom predikata prvoga reda (FOLTL)
4. Proširenje s prošlim vremenom (ptLTL) - "past time LTL" – ima istu izražajnu moć kao i LTL sa samo budućim vremenom

…

Usporedba CTL-a i LTL-a (1/3)

1. **Različita i neusporediva izražajnost** (ekspresivnost).
2. CTL **eksplicitno kvantificira putove**
3. LTL može selektirati **sve pojedinačne putove iz nekog stanja** koji zadovoljavaju LTL-formulu
4. Neke formule u CTL-u nije moguće izraziti u LTL-u i obratno.
5. U LTL-u **su složenije procedure provjere modela** ali jednostavnije neke druge procedure (npr. provjera valjanosti formula).

Usporedba CTL-a i LTL-a (2/3)

CTL-formula AG (EF *p*) = "iz kojeg god stanja da krenemo možemo doći (postoji barem jedan put) do stanja gdje je p istinit" **nema ekvivalenta u LTL-u**.

Primjer: Neka je φ LTL-formula takva da je A[φ] navodno ekvivalentno AG(EF *p*) , tj. za sve putove.

Za model kao na slici:

φ kao LTL-formula GF p ne vrijedi za put (beskonačnu petlju) stalno u s_0 (izdvojen put) jer mora vrijediti za sve pojedinačne putove.

φ = AG (EF *p*) kao CTL-formula vrijedi jer EF *p* vrijedi za oba stanja.

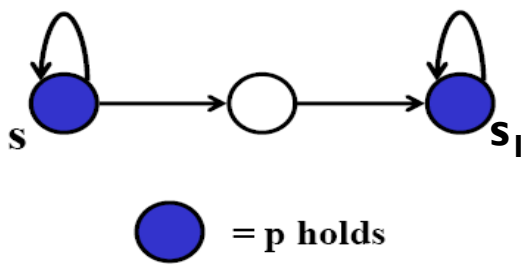
Usporedba CTL-a i LTL-a (3/3)

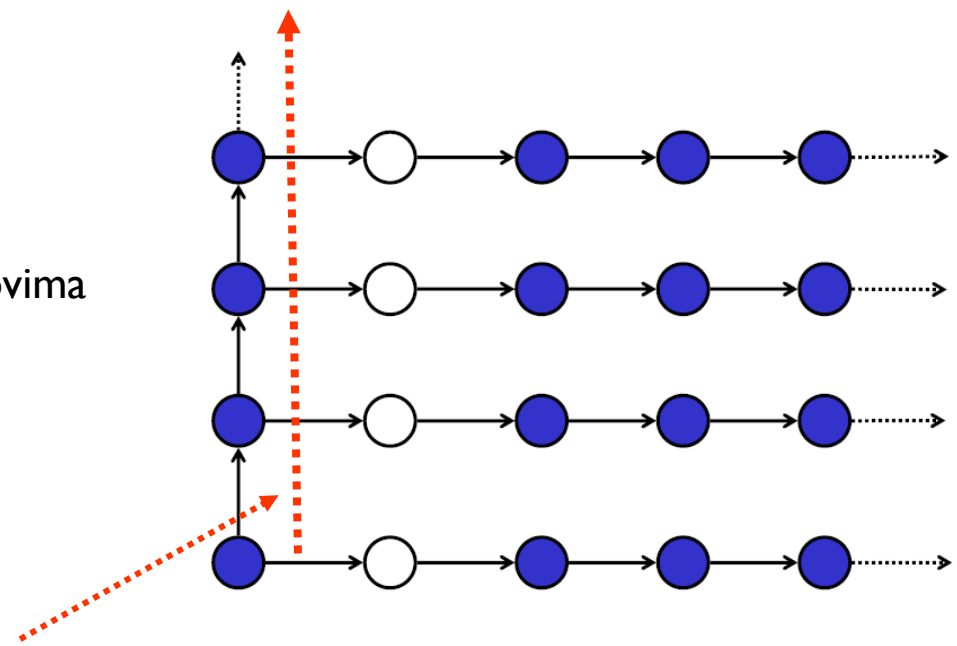
LTL-formula FG p nije ekvivalentna CTL-formuli AF (AG p)

FG p = konačno (nakon konačnog broja koraka) globalno (stalno) p

AF (AG p) = na svim putovima uvijek dolazimo konačno do stanja iz kojega je dalje stalno p=T.

U prikazanom modelu FG p vrijedi za sva stanja, a AF (AG p) ne vrijedi.





FG p – na svim **pojedinačnim** putovima dolazimo bilo kako do stanja nakon kojega je stalno p=TRUE.

AF (AG p) je striktno jači zahtjev.

AG p vrijedi samo za S_1.

Iz S postoji put natrag na S i nikad ne dođe u S_1.

Zadaci

1. Preslikajte rečenice prirodnog jezika u formule logike LTL:

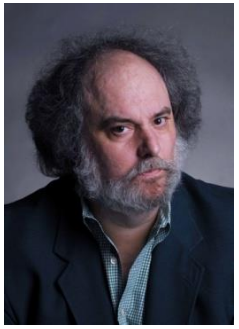
   a) "Nije moguće doći u stanje gdje vrijedi p i ne vrijedi q."

   b) "Uvijek se konačno dolazi u stanje gdje vrijedi p, a od idućeg stanja p vrijedi dalje beskonačno često."

   c) "Uvijek ako vrijedi p, q će vrijediti od tog stanja sve dok p ne prestane vrijediti.“

Potpuna logika vremenskog grananja: CTL*

E. A. Emerson i J. Y. Halpern (1986.)



CTL*

Unificirajuća struktura za CTL i LTL.

Dozvoljeno: 1) Booleove kombinacije F, G, X, U
2) Ugniježđivanje prije primjene E, A

Primjer 1: $A\,[(p\ U\ r) \vee (q\ U\ r)]$

"Duž svih putova p je istinit do r, ili q je istinit do r." Nije CTL, a za LTL samo ako vrijedi za sve pojedinačne putove.

Primjer 2: $E\,(GF\ p)$

"Postoji put na kojem je p istinit beskonačno često." Nije ni CTL ni LTL.

Primjer3: $A\,[X\ p \vee XX\ p]$

"Duž svih putova, p je istinit u sljedećem stanju ili u prvom stanju nakon sljedećeg." Nema ekvivalenta u CTL. U LTL-u samo ako vrijedi za sve pojedinačne putove.

CTL* - definicija sintakse

Formula stanja (dobro formirana CTL* formula):

φ: $p \mid \neg p \mid p \wedge q \mid p \vee q \mid p \Rightarrow q \mid p \Leftrightarrow q \mid E\,\phi \mid A\,\phi$

Formula puta:

ϕ: $\varphi \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid X\,\phi \mid F\,\phi \mid G\,\phi \mid \phi\,U\,\phi$

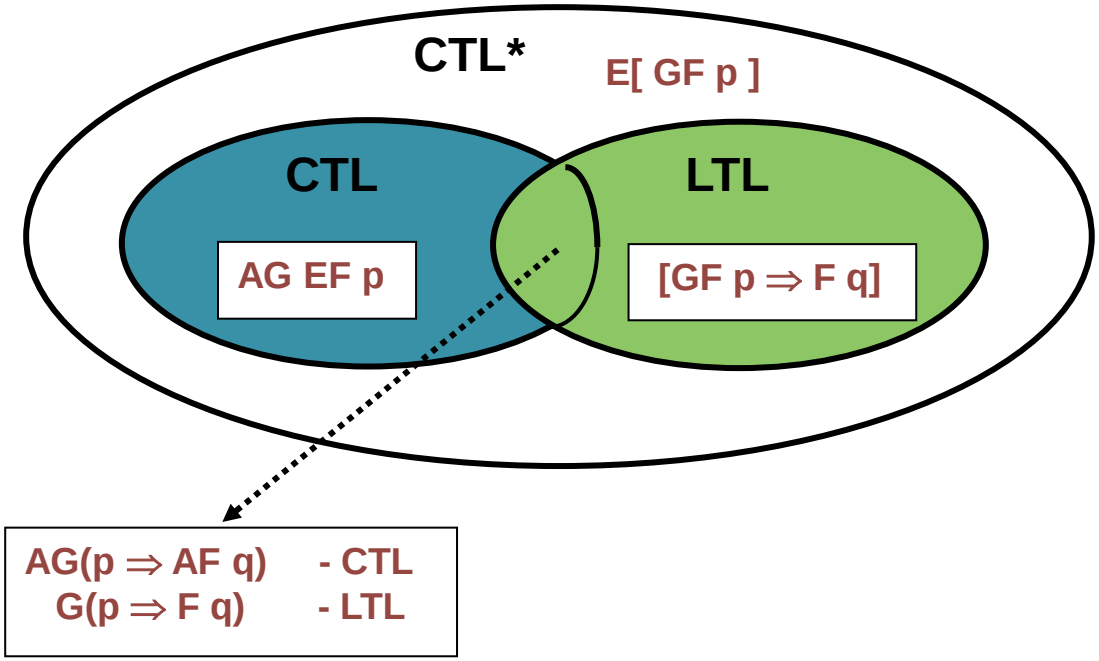
CTL je restriktivni CTL* za formulu puta:

ϕ: Ako su p, q formule stanja, tada su X p, F p, G p, p U q formule puta.

LTL form. stanja φ je ekvivalentna CTL* form. stanja A[φ], dok E[φ] nije dozvoljen, a formula puta je:

ϕ: $p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid X\,\phi \mid F\,\phi \mid G\,\phi \mid \phi\,U\,\phi$

CTL* - odnosi između vrem. logika



Objašnjenja formula sa slike

AG EF p	Iz svakog stanja možemo doći u stanje gdje je p=T. CTL, ali nije u LTL (ranije pokazano).
[GF p $\Rightarrow$ F q]	Ako na svim putovima vrijedi beskonačno često p=T, onda vrijedi i konačno q=T. Npr. beskonačno česti REQ implicira konačni ACK. LTL, ali ne CTL (pravednost se ne može izravno izraziti u CTL-u, a može u LTL-u). Nije isto kao AG AF p $\Rightarrow$ AF q (ako p vrijedi beskonačno često, onda q vrijedi konačno)
E[GF p]	Postoji put s beskonačno često p=T CTL*, ali nije u CTL (dokaz o nemogućnosti izražavanja ovakvog svojstva je složen), nije LTL jer je LTL ekvivalentan CTL* formuli A[φ], a ovdje je egzistencijski operator
AG (p $\Rightarrow$ AF q)	CTL, svaki p=T će konačno slijediti sa q=T
G (p $\Rightarrow$ F q)	LTL, ako vrijedi u strukturi M za svaki put posebno

Složenost provjere modela

Logika	Dokazana složenost provjere modela u odnosu na $\|\varphi\|$*
LTL	PSpace-Complete
CTL	P-Complete
CTL*	PSpace-Complete

Složenost provjere modela u sve tri logike je **linearna** u odnosu na **broj stanja i prijelaza Kripkeove strukture M**.
Problem je što je **broj stanja eksponencijalan** (ili još gori) u ovisnosti o broju varijabli

Poznato je da vrijedi:

$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME \subseteq NEXPTIME \subseteq EXPSPACE \subseteq 2\text{-}EXPTIME \subseteq ELEMENTARY$

* $|\varphi|$ - broj povezujućih elemenata (npr. za CTL to su logički operatori i operatori stanja AF, EU, EX...) koje čine formulu φ

Eksplicitni postupci računanja skupova stanja koja zadovoljavaju CTL-specifikaciju

Definicija problema

- Za danu Kripkeovu strukturu (usmjereni označeni graf) i određen skup početnih stanja S_0, provjeri da je CTL formula zadovoljena za ta stanja:

Formalno:

- $M, S_0 \models \phi$, tj. $\forall s_0 \in S_0 \quad M, s_0 \models \phi$

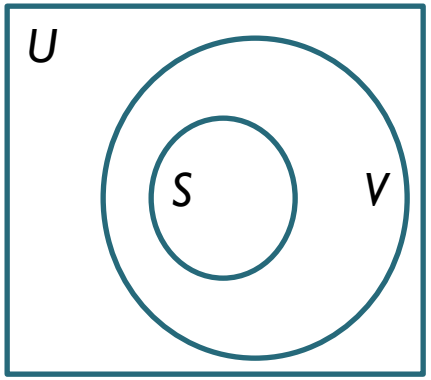
- Uobičajeni pristup: potrebno je pronaći sva stanja koja zadovoljavaju CTL formulu ϕ i ispitati je li željeni podskup S_0 uključen.
- Problem: učinkovit algoritam izračunavanja stanja

Postupci izračunavanja skupova stanja u verifikaciji sustava provjerom modela

- **Eksplicitno** predstavljanje i izračunavanje skupova stanja
  - Kripkeova struktura predstavljena je u memoriji računala kao označeni usmjereni graf (engl. *labeled directed graph*). Izračunavanje skupova stanja koji zadovoljavaju formulu vremenske logike izvodi se postupkom "čvrste točke".
- **Simbolički postupci** predstavljanja i izračunavanja skupova stanja
  - Skupovi stanja i relacija zadane Kripkeove strukture predstavljeni su Booleovim (logičkim) formulama. Booleove formule se u drugom koraku učinkovito predstavljaju binarnim dijagramima odlučivanja (BDD). Izračunavanje skupova stanja koja zadovoljavaju formulu vremenske logike također se izvodi postupkom "čvrste točke".

Notacija

- $v \in V$ Element v je član skupa V
- $v \notin V$ Element v nije član skupa V
- $|V|$ Kardinalnost (broj elemenata) skupa V
- $S \subseteq V$ Skup S je podskup skupa V
- $\varnothing$ Prazan skup (član svih skupova)
- S' Komplement skupa S
- U Svemir – nadskup svih skupova; vrijedi: $S'=U - S$



Notacija

- Partitivni skup skupa V (engl. *power set*):

To je skup skupova $\{S\}$ takvih da je svaki S podskup skupa V. Osim oznake 2^V za sve podskupove skupa V, često se koristi i oznaka **P(*V*).**

$P(V) = 2^V = \{S \mid S \subseteq V\}$

$|P(V)| = |2^V| = 2^{|V|}$ Kardinalnost partitivnog skupa od skupa V je potencija broja 2

Primjer: $V = \{1, 2, 3\}$

$P(V) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$

Relacija kao skup

Binarna relacija **na skupu stanja S:** $R \subseteq S \times S$
je Kartezijev produkt; daje skup uređenih parova elemenata skupa *S*.

Totalna binarna relacija Kripkeove strukture i uređenost:
Za svaki element *s* (u našem kontekstu "stanje sustava") iz skupa *S* postoji barem jedan (može i više) element *t* (u našem kontekstu "stanje sustava") takav da su elementi (stanja) *s* i *t* povezani relacijom *R*:

$\forall s \in S \ \{\exists t \in S \mid (s,t) \in R\}$ Kripke: svaki $s \in S$ je obuhvaćen u *R*.

Skup svih stanja {*t*} čine stanja dosezljiva (engl. *reachable*) u jednom koraku iz skupa stanja {s}.
Totalna binarna relacija *R* na Kripkeovoj strukturi je skup svih mogućih tranzicija (prijelaza) između stanja.

Definicije

$$\forall s \in S \ \{\exists t \in S \mid (s,t) \in R\}$$

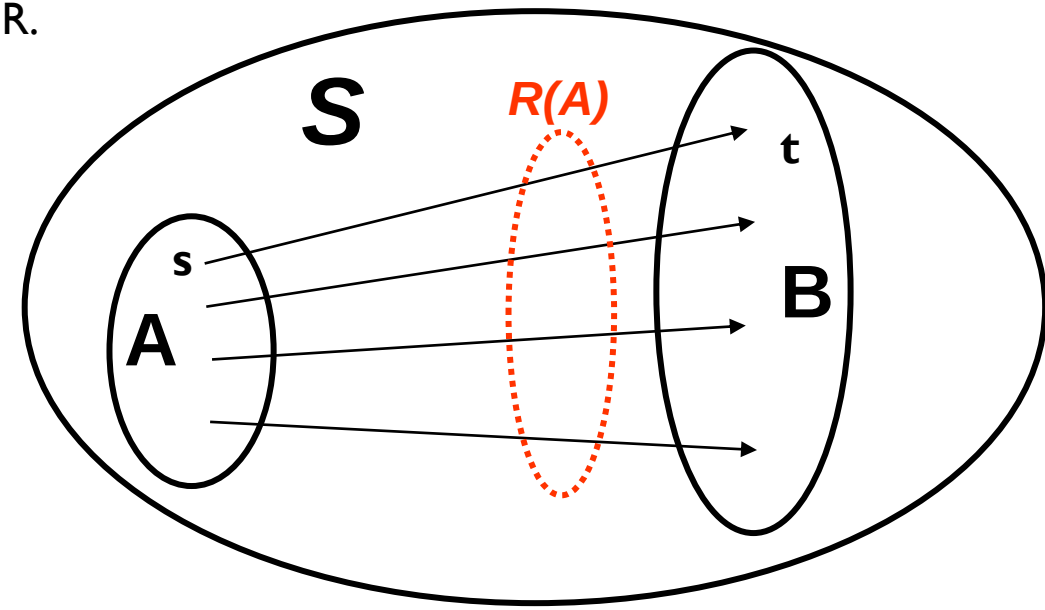
Skup $\{t\}$ je **slika (engl. *IMAGE*)** skupa $\{s\}$ pod relacijom R.

Skup $\{s\}$ je **pred-slika (engl. *PRE-IMAGE*)** skupa $\{t\}$ pod relacijom R.

Skup $\{s\}$ je **slika (engl. *IMAGE*)** skupa $\{t\}$ pod relacijom R^{-1} (inverznom).

Inverzna relacija

- Neka na skupu S postoji relacija $R \subseteq A \times B$, gdje je: $A = \{s\}$, $B = \{t\}$.
- A = slika (IMAGE) pod **inverznom relacijom** R^{-1}
- B = slika (IMAGE) od A pod R.
- **$A = R^{-1}(B)$**
- **$B = R(A)$**



- $R(s) = \{t \in S \mid (s,t) \in R\}$
- $R^{-1}(t) = \{s \in S \mid (s,t) \in R\}$
- Primjena inverzne relacija R^{-1} na stanje t daje jedno ili više stanja s iz kojih u jednom koraku dolazimo do specificiranog stanja t.
- $R^{-1}(T) = \cup_{t \in T} R^{-1}(t)$ ako je T skup $\{t\}$, rezultat je unija svih $\{s\}$.

<u>Za Kripkeovu strukturu</u>: $R^{-1}(S) = S$ prethodnici svih stanja su sva stanja (R je totalna relacija).

$R^{-1}(\emptyset) = \emptyset$ nema prethodnika praznog skupa

Izračunavanje slike

- Postupci izračunavanja slike ili pred-slike preko relacije R, ili preko inverzne relacije R^{-1}, predstavljaju najznačajniji dio analize dosezljivih stanja (engl. *reachability analysis*) u sustavima s prijelazima (engl. *transition systems*).
- Neki postupci temelje se na izravnom (eksplicitnom) izračunavanju stanja.
- Neki drugi postupci uvode transformacije preko logičkog kodiranja, pa se slika računa u transformiranom prostoru i zatim dekodira – algoritam za izračun dosezljivih stanja u tom slučaju ćemo pokazati kasnije
- Pretpostavljamo da postoji algoritam izračuna $R(s)$ i $R^{-1}(t)$.

Logičke rekurzije CTL-formula

$AG\ \varphi = \varphi \wedge AX\ AG\ \varphi$; sada i na svim putovima
; počevši od sljedećeg

$EG\ \varphi = \varphi \wedge EX\ EG\ \varphi$; sada i na jednom putu
; počevši od sljedećeg

$AF\ \varphi = \varphi \vee AX\ AF\ \varphi$; sada ili za svako sljedeće
; stanje vrijedi $AF\ \varphi$

$EF\ \varphi = \varphi \vee EX\ EF\ \varphi$; sada ili za jedno sljedeće
; stanje vrijedi $EF\ \varphi$

$A[\varphi\ U\ \psi] = \psi \vee (\varphi \wedge AX\ A(\varphi\ U\ \psi))$
; ψ vrijedi sada ili
; φ vrijedi sada i za svako
; sljedeće stanje vrijedi
; $A(\varphi\ U\ \psi)$

$E[\varphi\ U\ \psi] = \psi \vee (\varphi \wedge EX\ E(\varphi\ U\ \psi))$; slično kao AU, ali
; samo za jedan put

Izračunavanje EX, EG, EU (adekvatan skup) omogućuje izračunavanje svih CTL-formula.

Zamjena logičkih operacija operacijama nad skupovima

Model M = (S, R, L) EX, EG, EU - adekvatan skup

$R(s) = \{ t \in S \mid (s, t) \in R \}$ - daje sljedbenike stanja *s* (skup *t*-ova)

$Q(\text{False}) = \varnothing$
$Q(\text{True}) = S$
$Q(p) = \{ s \mid p \in L(s) \}$ - skup stanja s u kojima vrijedi p = True
$Q(\neg f) = S - Q(f)$ - sva stanja u S osim onih u kojima f = True
$Q(f \wedge g) = Q(f) \cap Q(g)$ - skup dobiven presjekom skupova

$Q(EX\, f) = \{ s \mid R(s) \cap Q(f) \}$ - stanja koja imaju sljedbenike u Q(f)
(R(s) daje sve sljedbenike u jednom koraku)

$Q(EG\, f) = Q(f) \cap Q(EX\, EG\, f)$ - stanja Q(f) u kojima je f = True
i stanja za koja vrijedi Q(EG f)
nakon jednog koraka (EX)

$Q[E(f\, U\, g)] = Q(g) \cup [Q(f) \cap Q(EX\, E(f\, U\, g))]$ - stanja u kojima je g = True, ili
stanja u kojima je f = True i nakon jednog koraka
(EX) vrijedi E(f U g)

Izračunavanje skupa stanja za CTL-formulu EX

Zadan je skup stanja Q(*f*) u kojima je istinita formula vremenske logike *f*. Potrebno je pronaći skup stanja **Q(EX *f*)**, dakle ona stanja iz kojih u jednom koraku dolazimo do nekog stanja iz Q(*f*).

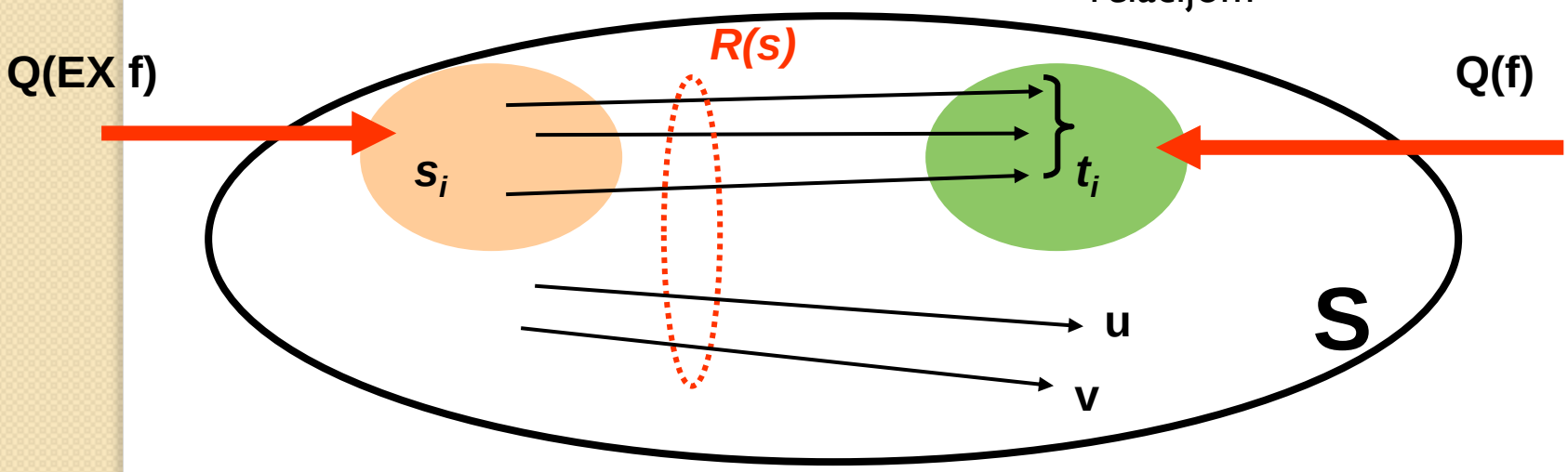
$$Q(\,EX\ f\,) = \{\, s \mid R\,(\,s\,) \cap Q(\,f\,) \neq \varnothing \,\}$$

$$= \{\, s \mid \exists_{t \in R\,(\,s\,)}\ t \in Q(\,f\,)\}$$

$$= \{\, s \mid \exists_{t \in Q\,(\,f\,)}\ (s, t) \in R\}$$

$$\boxed{= R^{-1}\,(\,Q\,(\,f\,))}$$

Slika pod inverznom relacijom



CTL-operatori kao skupovi stanja

1. Izračunavanje skupa **Q(EX f)** (pokazano ranije):

$$Q(EX\ f) = R^{-1}\ (Q(f))$$

2. Izračunavanje skupa **Q(EG f)** je uz supstituciju za EX:

$$Q(\ EG\ f\ ) = Q(f\ ) \cap Q(\ EX\ EG\ f\ )$$

$$Q(\ EG\ f\ ) = Q(f\ ) \cap R^{-1}\ (Q(\ EG\ f\ ))$$

3. Izračunavanje skupa **Q(E (f U g))** je uz supstituciju za EX:

$$Q[\ E(\ f\ U\ g)] = Q(\ g\ ) \cup [Q(\ f\ ) \cap Q(\ EX\ E(\ f\ U\ g))]$$

$$Q[\ E(\ f\ U\ g)] = Q(\ g\ ) \cup [Q(\ f\ ) \cap R^{-1}\ (Q(\ E(\ f\ U\ g)))]$$

Za izračunavanje 2. i 3. potrebna je teorija "čvrste točke" jer nije očigledno kako razriješiti rekurzije.

Fiksna (čvrsta) točka (1/6)

- Monotone funkcije i fiksna točka

<u>Definicije:</u>

S	- skup stanja
F: P(S) $\rightarrow$ P(S)	- funkcija na svim podskupovima u S,
P(S)	- 2^S (partitivni skup)

1. F je monotona akko
   X $\subseteq$ Y implicira (povlači) F(X) $\subseteq$ F(Y)
   za sve podskupove X i Y u S (oznaka $\subseteq$ - podskup).

2. Podskup X od skupa S je fiksna točka (engl. *fix-point*) funkcije F akko:

$$\mathbf{F(X)=X}$$

Fiksna (čvrsta) točka (2/6)

<u>Primjer 1:</u>
Neka je $S = \{s_0, s_1\}$,
te neka je $F(Y) = Y \cup \{s_0\}$ za sve podskupove $Y \subseteq S$.

Test na monotonost:
Neka je Y' također bilo koji podskup od S.
Svaki $Y' \subseteq Y$, implicira $Y' \cup \{s_0\} \subseteq Y \cup \{s_0\}$, te je F monotona.

Analiza fiksne točke (za sve podskupove $Y \subseteq S=\{s_0, s_1\}$):
Podskup { } nije fiksna točka jer $F(\{\ \}) = \{\ \} \cup \{s_0\} = \{s_0\}$.
Podskup $\{s_0\}$ je najmanji *fix-point*, jer $F(\{s_0\})=\{s_0\} \cup \{s_0\}=\{s_0\}$.
Podskup $\{s_1\}$ nije fiksna točka jer $F(\{s_1\}) = \{s_1\} \cup \{s_0\} = \{s_0, s_1\}$.
Skup $\{s_0, s_1\}$ je najveći *fix-point*, jer $F(\{s_0, s_1\})=\{s_0, s_1\}\cup\{s_0\}=\{s_0, s_1\}$.

Monotone funkcije uvijek imaju najmanju i najveću fiksnu točku.

Funkcije za izračunavanje skupova stanja u Kripkeovoj strukturi koje nas zanimaju su monotone te imaju najmanji i najveći *fix-point*:

$Q(\ EG\ f\ ) = Q(f) \cap Q(\ EX\ EG\ f\ )$
$Q(\ E(f\ U\ g)\ ) = Q(g) \cup [Q(f) \cap Q(\ EX\ E(\ f\ U\ g\ ))]$

Fiksna (čvrsta) točka (3/6)

Primjer 2:
$S = \{s_0, s_1\}$
Funkcija: $G(Y)$ = ako $[Y = \{s_0\}]$ tada $\{s_1\}$ inače $\{s_0\}$

Test na monotonost:
Primjena funkcije G na $Y = \{s_0, s_1\}$ daje $\{s_0\}$.
Primjena funkcije G na $Y' = \{s_0\}$ daje $\{s_1\}$.
Y' je podskup od Y, tj. $(Y' \subseteq Y)$,
ali kako rezultat $\{s_1\}$ nije podskup od $\{s_0\}$ to G nije monotona.

Analiza fiksne točke (za sve podskupove $Y \subseteq S=\{s_0, s_1\}$):
$G(\{\}) = \{s_0\}$
$G(\{s_0\}) = \{s_1\}$
$G(\{s_1\}) = \{s_0\}$
$G(\{s_0, s_1\} = \{s_0\}$

G(Y) nema nijednu fiksnu točku. **Nemonotone funkcije mogu, ali i ne moraju imati fiksnu točku.**

Fiksna (čvrsta) točka (4/6)

Postupak izračunavanja fiksne točke: Teorem Knaster-Tarski

Neka je S skup: $S = \{s_0, s_1, \ldots, s_n\}$ sa n+1 elementom.

Označimo sa F^i : funkcija F primijenjena i-puta, odnosno: $F(F(\ldots F(X)))$

Npr. Neka je $F(Y) = F^1(Y) = Y \cup \{s_0\}$ gdje je $Y \subseteq S$

$F^2(Y) = F(F(Y)) = [Y \cup \{s_0\}] \cup \{s_0\} = Y \cup \{s_0\} = F(Y)$, te je $F^2 = F$

Za ovaj primjer vrijedi: $F^i = F$ za sve $i \geq 1$

Teorem [P(S) je partitivni skup]:

Ako je $F: P(S) \rightarrow P(S)$ **monotona**, tada

$F^{n+1}(\emptyset)$ je najmanji *fix-point* od F.

$F^{n+1}(S)$ je najveći *fix-point* od F.

Fiksna (čvrsta) točka (5/6)

Dokaz da je $F^{n+1}(\emptyset)$ najmanji *fix-point* od F: (napomena: dokaz nije potrebno učiti)

F je monotona (uvjet) pa vrijedi $\emptyset \subseteq F(\emptyset)$, također $F(\emptyset) \subseteq F(F(\emptyset))$, odnosno $F^1(\emptyset) \subseteq F^2(\emptyset)$.
Indukcijom slijedi:
$F^1(\emptyset) \subseteq F^2(\emptyset) \subseteq \ldots \subseteq F^i(\emptyset)$ za sve $i \geq 1$

Definiramo: **$i = n + 1$** (**$n + 1$** = broj elemenata u skupu S).
Tvrdimo: jedan od gornjih $F^k(\emptyset)$ je *fix-point*, tj. $F(F^k(\emptyset))=F^k(\emptyset)$

Kad $F^1(\emptyset)$ ne bi bio *fix-point* onda bi $F^1(\emptyset)$ morao sadržavati najmanje 1 element više od $\emptyset$ (jer tada $\emptyset \neq F(\emptyset)$).
$F^2(\emptyset)$ bi morao sadržavati barem 2 elementa, morao biti veći od $F^1(\emptyset)$.
Svaki daljnji bi morao imati barem jedan element više od prethodnika.

Kad $F^{n+1}(\emptyset)$ ne bi bio *fix-point*, $F^{n+2}(\emptyset) = F(F^{n+1}(\emptyset))$ bi morao imati $n+1+1$ element, što je nemoguće jer S ima samo $n+1$ elemenata. Dakle $F^{n+1}(\emptyset)$ mora biti *fix-point*.

Odnosno: $F(F^{n+1}(\emptyset)) = F^{n+1}(\emptyset)$ $F^{n+1}(\emptyset)$ je fiksna točka

Fiksna (čvrsta) točka (6/6)

Još treba dokazati da je to najmanja fiksna točka!

Neka je X neki drugi *fix-point* od F, tj. $F(X) = X$
Moramo pokazati da je $F^{n+1}(\emptyset) \subseteq X$.

Kako je $\emptyset \subseteq X$, to slijedi $F(\emptyset) \subseteq F(X) = X$ (jer je funkcija monotona)
Dakle: $F(\emptyset) \subseteq X$.
$F^2(\emptyset) \subseteq F(F(X)) = X$ (jer je X fix-point)
Indukcijom $F^i(\emptyset) \subseteq X$ za sve $i \geq 0$, pa i za $i = n + 1$, slijedi $F^{n+1}(\emptyset) \subseteq X$

Dokaz za najveći fix-point analogno uz zamjenu: $\subseteq$ sa $\supseteq$, te $\emptyset$ sa S.

Teorem daje ujedno i algoritam izračunavanja i garantira završetak:

<u>Najmanji *fix-point*</u>: iterativna primjena F na prazan skup $\emptyset$, dok rezultat ne postane invarijantan na tu primjenu.

<u>Najveći fix-point</u>: iterativna primjena F na skup svih stanja S, dok rezultat ne postane invarijantan na tu primjenu.

Najveća gornja granica broja iteracija: n+1 (za S sa n+1 elementom)

Izračunavanje EG preko najveće fiksne točke (1/3)

- $Q(\ EG\ f\ ) = Q(\ f\ ) \cap Q(\ EX\ EG\ f\ )$
  $= Q(\ f\ ) \cap R^{-1}(Q(EG\ f))$

- Primjenom funkcije $F(X) = Q(\ f\ ) \cap R^{-1}(X)$ i to n+1 puta na skup svih stanja S slijedi najveća čvrsta točka, tj. podskup Z_{EG} koji zadovoljava CTL-formulu EG f:

  $F^{n+1}(S) = Q(EG(f))$

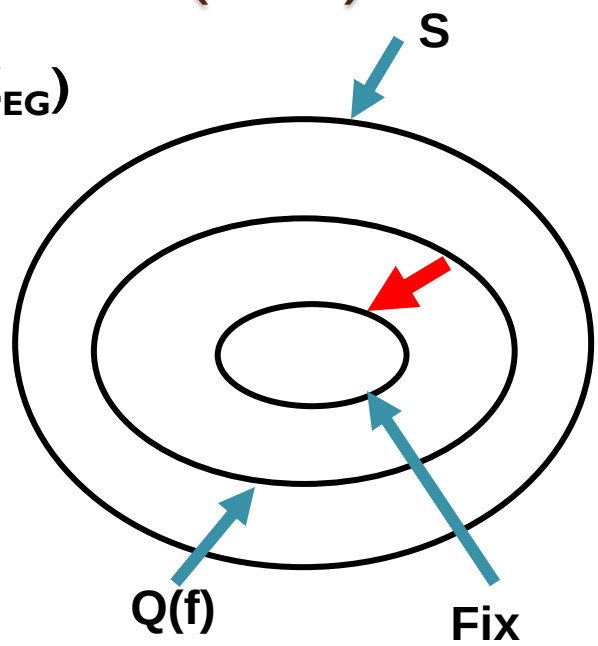
- $Z_{EG} = F_{EG}\ (Z_{EG}) = Q(\ f\ ) \cap R^{-1}(Z_{EG})$

  F_{EG}

Izračunavanje EG preko najveće fiksne točke (2/3)

Q(EG f): $Z_{EG} = F_{EG}(Z_{EG}) = Q_f \cap R^{-1}(Z_{EG})$

```
Q(EG f) (Q(f)): {
        k := 0;    Z_k := S;
        do         {
                   Z_{k+1} := Q(f) ∩ R^{-1} (Z_k);
                   if (Z_{k+1} = Z_k ) return Z_k ;
                   k++ ;
        } forever;
}
```



Započinjemo sa skupom S, tj. $Z_0 = S$, i prva iteracija daje:
$Z_1 = Q(f) \cap R^{-1}(S) = Q(f)$, dakle u prvoj iteraciji je $Z_1 \neq Z_0$ te se ide dalje:
$Z_2 = Q(f) \cap R^{-1}(Q(f))$ … itd. sve dok $Z_{n+1} = Z_n$ tj. dosegne fiksnu točku

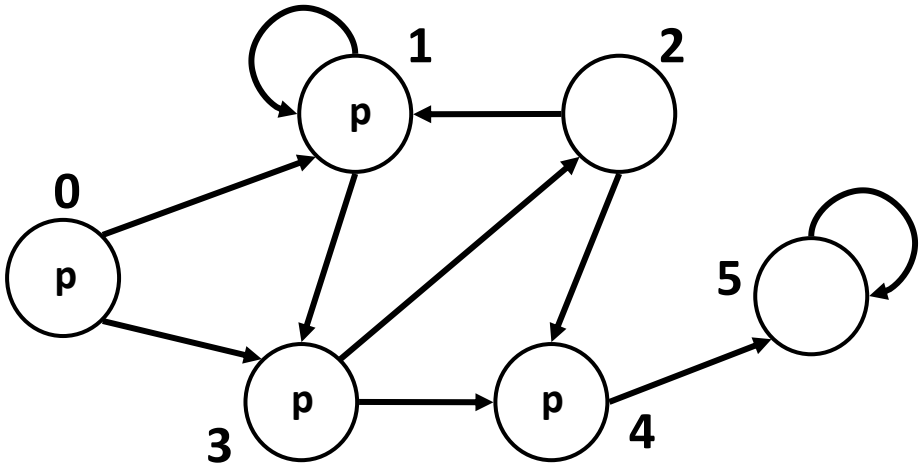
Budući da $R^{-1}(S) = S$, bolje je odmah započeti sa $Z_k = Q(f)$.

Izračunavanje EG preko najveće fiksne točke (3/3)

Za primjer sa slike odredi stanja za koja vrijedi EG p:

$Q(p) = \{ 0, 1, 3, 4 \}$, tu je p=True

$$\boxed{Z_{k+1} = Q(p) \cap R^{-1}(Z_k)}$$



Početno: $Z_0 = S = \{ 0, 1, 2, 3, 4, 5 \}$

- $R^{-1}(Z_0) = R^{-1}(S) = S$ $= \{ 0, 1, 2, 3, 4, 5 \}$
- $Z_1 = \{ 0, 1, 3, 4 \} \cap R^{-1}(\{ 0, 1, 2, 3, 4, 5 \})$ $= \{ 0, 1, 3, 4 \}$ $\neq Z_0$
- $R^{-1}(Z_1) = R^{-1}(\{ 0, 1, 3, 4 \})$ = prethodnici $= \{ 0, 1, 2, 3 \}$
- $Z_2 = \{ 0, 1, 3, 4 \} \cap R^{-1}(Z_1)$ $= \{ 0, 1, 3 \}$ $\neq Z_1$
- $R^{-1}(Z_2) = R^{-1}(\{ 0, 1, 3 \})$ = prethodnici $= \{ 0, 1, 2 \}$
- $Z_3 = \{ 0, 1, 3, 4 \} \cap R^{-1}(Z_2)$ $= \{ 0, 1 \}$ $\neq Z_2$
- $R^{-1}(Z_3) = R^{-1}(\{ 0, 1 \})$ = prethodnici $= \{ 0, 1, 2 \}$
- $Z_4 = \{ 0, 1, 3, 4 \} \cap R^{-1}(Z_3)$ $= \{ 0, 1 \} = Z_3$ (fiksna točka)

Rješenje: stanja {0, 1} zadovoljavaju EG p.

Izračunavanje EU putem najmanje fiksne točke (1/2)

$Q[\, f\ EU\ g] = Q(\, g\,) \cup [Q(\, f\,) \cap Q(\, EX\ E(\, f\ U\ g))]$

$Q[\, E(\, f\ U\ g)] = Q(\, g\,) \cup [Q(\, f\,) \cap R^{-1}\ (Q(\, E(\, f\ U\ g))]$

Primjenom funkcije $F(X) = Q(\, g\,) \cup [Q(\, f\,) \cap R^{-1}(X)]$ i to n+1 puta na prazan skup $\emptyset$ slijedi najmanja čvrsta točka, tj. podskup Z_{EU} koji zadovoljava CTL formulu E(f U g):

$F^{n+1}(\emptyset) = Q(E\ (f\ U\ g))$

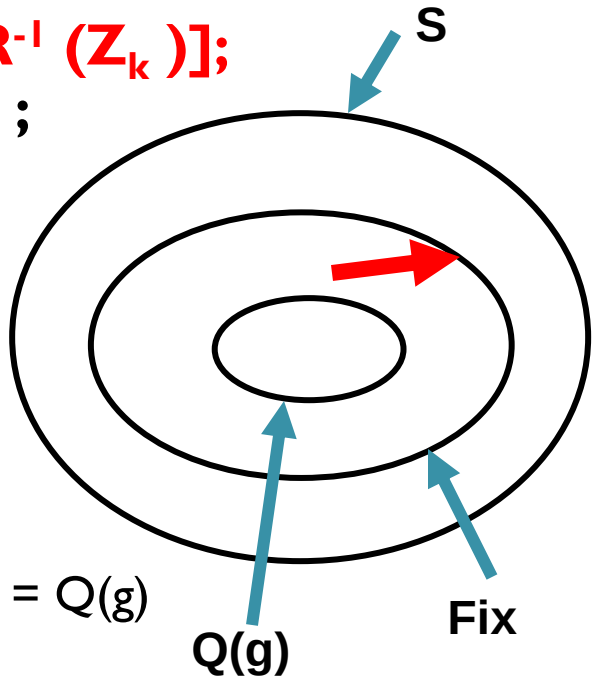
- $Z_{EU} = F_{EU}(Z_{EU}) = Q(\, g\,) \cup Q(\, f\,) \cap R^{-1}(Z_{EU})$

F_{EU}

Izračunavanje EU putem najmanje fiksne točke (2/2)

Q(f EU g): $Z_{EU} = F_{EU}(Z_{EG}) = Q_g \cup [Q_f \cap R^{-1}(Z_{EU})]$

```
Q(f EU g) (Q(f), Q(g))        {
        k := 0;   Z_k := ∅;
        do        {
                  Z_{k+1} := Q(g) ∪ [Q(f) ∩ R^{-1} (Z_k )];
                  if (Z_{k+1} = Z_k)   return Z_k ;
                  k++ ;
        } forever;
}
```

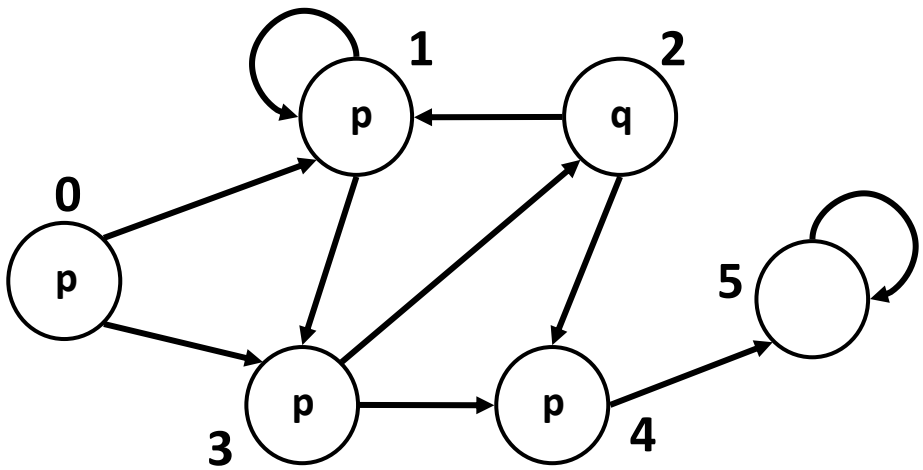


Započinjemo s praznim skupom $Z_0 = \emptyset$.

Budući da $R^{-1}(\emptyset) = \emptyset$, bolje odmah započeti sa $Z_k = Q(g)$

1. zadatak

- Za primjer sa slike odredite stanja za koja vrijedi E(p U q) korištenjem algoritma za izračunavanje EU pomoću najmanje fiksne točke.



2. zadatak

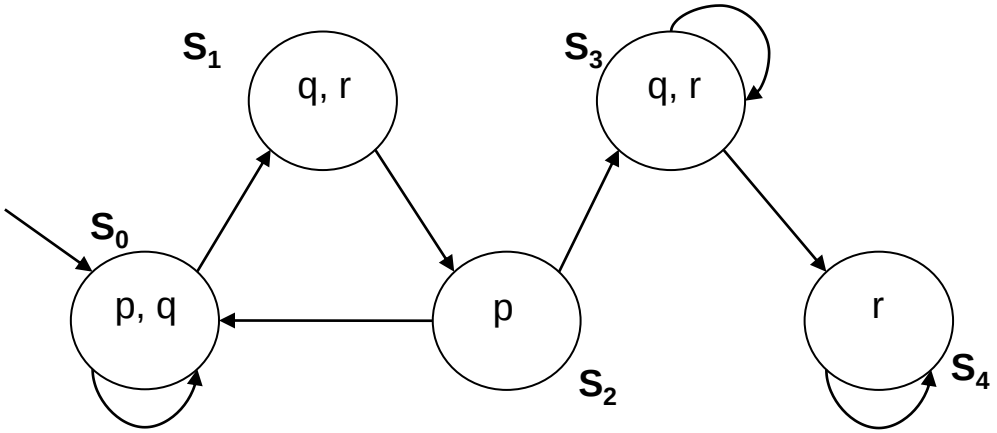
- Za funkciju $F(X) = (X \cup \{s1\}) \cap \{s2\}$ i skup mogućih stanja $S = \{s0, s1, s2\}$ odredite:

a) Je li funkcija monotona.

b) Ako funkcija jest monotona, nađite najmanju i najveću fiksnu točku.

Zadatak za bonus bod

- Koristeći teoriju fiksne točke i odgovarajući algoritam, odredite Q(EG r) za Kripkeovu strukturu prikazanu na slici. Potrebno je napisati cjelokupni postupak dobivanja rješenja i konačno rješenje.



FVPP: Java PathFinder

Primjer provjere modela za programe pisane u Javi

Pripremio: izv. prof. dr. sc. Alan Jović

Ak. god. 2022./2023.

Sadržaj

- O Java PathFinderu
- Struktura Java PathFindera
- Specifikacija svojstava programa
- Napredne teme i proširenja
- O 2. domaćoj zadaći

O JAVA PATHFINDERU

P. Mehlitz

N. Rungta

C. Pasareanu

W. Visser

Što je Java PathFinder?

- Java PathFinder (dalje: JPF) je radni okvir i skup alata za formalnu verifikaciju programa pisanih u programskom jeziku Java i to metodom provjere modela
- Vrlo složen radni okvir koji nudi niz mogućnosti proširenja
- Jedan od rijetkih uspješnih alata za izravnu verifikaciju izvornog koda programa (ne apstrakcije programa)
- Razvio ga je tim znanstvenika u NASA-i u suradnji s većim brojem sveučilišta, pojedinaca i poduzeća u svijetu (više desetaka takvih subjekata, od poduzeća se navodi Fujitsu)
- Alat otvorenog koda koji je distribuiran pod Apache 2.0 licencom
- Jezgra JPF-a je **virtualni stroj (VM) koji se izvodi iznad Javinog virtualnog stroja** kako bi omogućio provjeru modela korisničkih programa

Razvoj JPF-a

1999. - projekt je započeo razvojem JPF-a kao translatora podskupa Jave 1.0 u jezik PROMELA za alat SPIN koji se koristi za formalnu verifikaciju

2000. - reimplementacija sustava kao virtualnog stroja za provjeru modela (tipično otkrivanje kvarova pri istovremenom izvođenju)

2003. - uvođenje sučelja za proširenja

2005. - sustav postaje otvorenog koda na sjedištu SourceForge (http://javapathfinder.sourceforge.net/)

2008.-danas - poboljšanja koda sudjelovanjem na Google Summer of Code

https://summerofcode.withgoogle.com/programs/2022/organizations/the-jpf-team

2009. - sustav je prebačen na vlastiti poslužitelj koji podržava proširenja i Wiki stranice (http://babelfish.arc.nasa.gov/trac/jpf)

2018. - dio sustava je prebačen na Github te ga održava zajednica:

https://github.com/javapathfinder

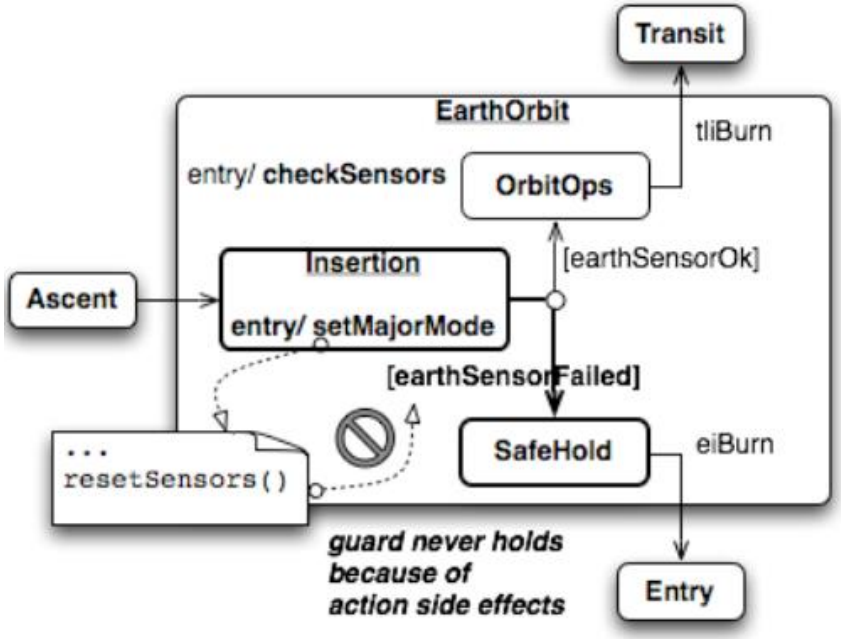
https://github.com/javapathfinder/jpf-core/wiki

2019. - poslužitelj na babelfish.arc.nasa.gov zatvoren za javnost

Zašto je razvijen JPF?

- **Pokrivenost linija koda** je nužno provjeriti kod kritičnih aplikacija – npr. lansiranja orbitera (NASA)
- Automatsko kodiranje UML-dijagrama stanja u program u Javi i isprobavanje može li se doći do svih dijelova koda – **analiza dosezljivosti**

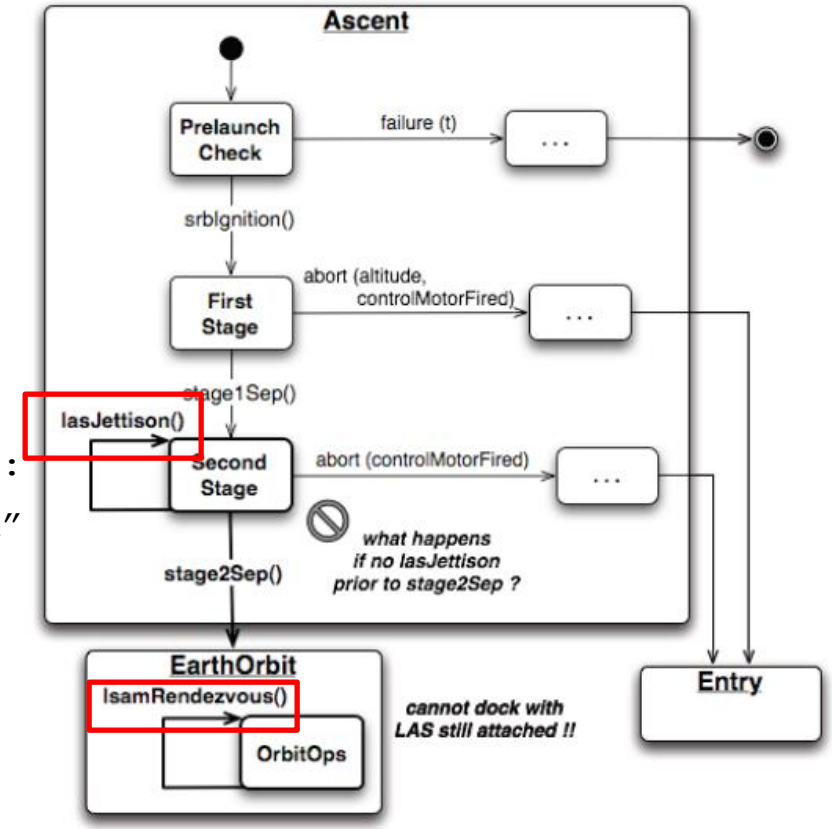
- U ovom primjeru, akcija `checkSensors()` isprobava senzore prilikom ulaska u stanje `EarthOrbit`, no varijabla vezana uz senzore (`earthSensorOk/Failed`) se provjerava tek kasnije – moguće je da senzore akcija `setMajorMode()` resetira tako da stanje `SafeHold` postane **nedosezljivo**



Zašto je razvijen JPF?

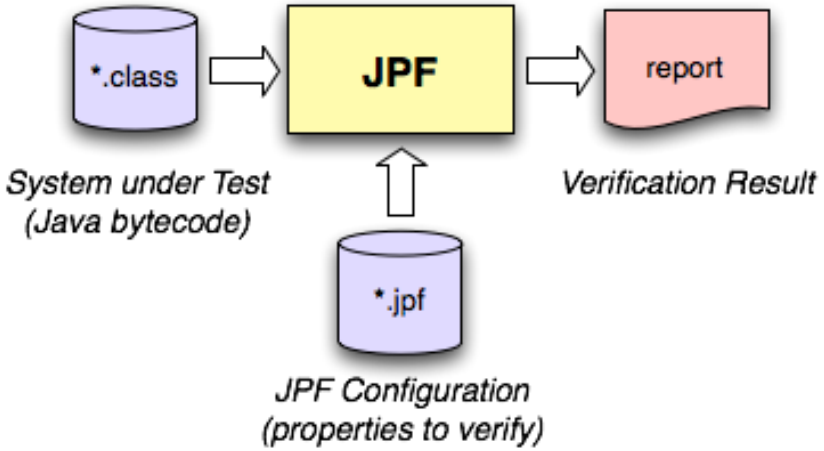
- Događaj `lasJetisson()` – odvajanje *launch abort* sustava (LAS) s letjelice
- `lsamRendevous()` – spajanje s drugom letjelicom – ono neće uspjeti ako se ne izvede `lasJetisson()` u stanju *Second Stage*, no to se ne osigurava u dijagramu
- Ascent i EarthOrbit module su možda čak radili različiti inženjeri
- Uvode se **tvrdnje** koje trebaju biti zadovoljene:

```
class OrbitOps {…
  void lsamRendezvous(){…
  assert !spacecraft.contains(LAS) :
  "lsamRendezvous with LAS attached"
  …
 } …
}
```



Što radi Java PathFinder?

- JPF radi provjeru modela nad **kompajliranim Javinim aplikacijama** – aplikacija koja se provjerava naziva se **"*System under Test*" (SUT)**
- Java aplikacije mogu biti pisane na tri načina:
  1. **neovisno o JPF-u** (najčešći slučaj)
  2. **s podrškom za JPF** – sadržavaju određene anotacije koje koristi JPF
  3. **ovisno o JPF-u** – pisane specifično za JPF (rjeđi slučaj)
- Specifikacija svojstava koja SUT treba zadovoljiti radi se kroz konfiguraciju JPF-a (skup konfiguracijskih datoteka)
- JPF generira izvještaj (engl. *report*) u kojem detaljno piše rezultat provjere modela



Kada koristiti JPF?

- JPF nije pogodan za korištenje nad sekvencijskim programima s malim brojem dobro definiranih ulaznih vrijednosti – ***unit-testovi* su prikladniji za te primjene!**
- JPF se koristi za:

1. Istraživanje alternativnih izvršavanja
   - **Sekvence raspoređivanja – konkurentne aplikacije** – najčešći razlog korištenja
   - **Varijacije u ulaznim podacima**
   - **Reakcije na događaje okoline**
   - **Izbor kontrolnog toka programa** – sustavno istraživanje strukture programa
2. Inspekciju izvršavanja programa
   - Implementacija analizatora prekrivanja koda ili specifičnih neinvazivnih testova koji mogu otkriti uvjete koje je inače teško otkriti (npr. preljev, podljev)

Što otkriva JPF?

- **Jezgra JPF-a** (`jpf-core` projekt) otkriva takozvana **"svojstva nefunkcioniranja"** (engl. *non-functional properties*)
  - **Svojstva koja se ne bi smjela ispoljiti ni u jednoj aplikaciji, a ona uključuju: potpuni zastoj, utrke, neuhvaćene iznimke, provjere tvrdnji (`assert`).**
- **Proširenja JPF-a** otkrivaju razna korisnički specificirana svojstva – uglavnom korištenjem **"slušača" (engl. *listener*)**
  - Dodaci koji omogućuju da se pozorno prate sve akcije koje radi JPF, npr. izvođenje određenih instrukcija, stvaranje objekata, dolaska do određenog stanja programa, itd.
- JPF omogućuje prikaz **kompletne povijesti izvršavanja – traga** (engl. *trace*), čak do razine svake instrukcije *bytecode*, koja je dovela do otkrivanja pogreške.

Primjer otkrivanja utrke

```
public class Racer implements Runnable {
     int d = 42;
     public void run () {
          doSomething(1001);
          d = 0;                          // (1)
     }

   public static void main (String[] args){
          Racer racer = new Racer();
          Thread t = new Thread(racer);
          t.start();
          doSomething(1000);
          int c = 420 / racer.d;          // (2)
          System.out.println(c);
     }

     static void doSomething (int n) {
          // not very interesting..
          try { Thread.sleep(n); }
          catch (InterruptedException ix) {}
     }
}
```

Dvije dretve pristupaju istoj varijabli `d`. Dijeljenje s nulom se događa ako druga dretva (ona stvorena iz izvorne unutar metode `main`) izvede naredbu (1) prije nego što prva dretva izvede naredbu (2). To je moguće zbog načina kako se dretve raspoređuju za izvođenje.

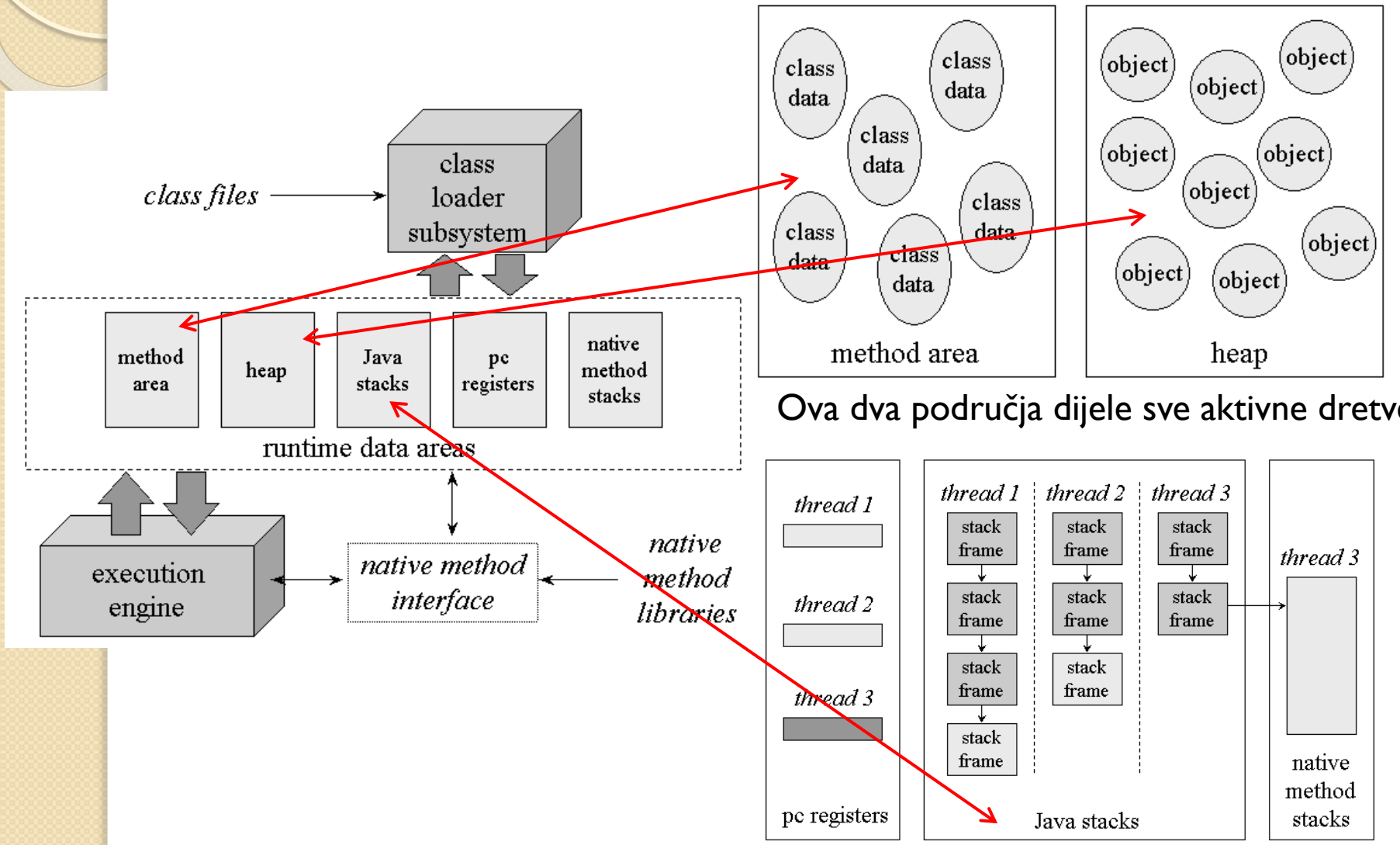
Bez ikakvog dodatnog podešavanja, JPF će pronaći neuhvaćenu iznimku `java.lang.ArithmeticException – division by zero`

Uz podešavanje (korištenje slušača `PreciseRaceDetector`), navest će točno retke koda (1) i (2) koji uzrokuju utrku dretvi.

Kako radi Java PathFinder?

- JPF je virtualni stroj koji izvršava naš SUT ne samo jednom (kao uobičajeni Javin VM), već na sve moguće načine
- Ako nađe mjesto kvara, onda izvještava o svakom koraku od početka do mjesta kvara
- U uobičajenom načinu izvršavanja (`jpf-core`), JPF je alat za provjeru modela s **eksplicitnim pamćenjem stanja izvođenja programa** (engl. *explicit state model checker*). To znači da pamti:
  - **okvire stoga dretvi (engl. *stack frames*)**
  - **objekte na gomili (engl. *heap objects*) i**
  - **stanja dretvi (engl. *thread states*)**
- Eksplicitno pamćenje stanja dovodi do **problema eksplozije broja stanja** (engl. *state explosion problem*)

Unutarnji pogled na ustroj Javinog virtualnog stroja (JVM)



Ova dva područja dijele sve aktivne dretve

Unutarnji pogled na ustroj Javinog virtualnog stroja (JVM)

- ***Class files*** – prevedeni kod – međukod (*bytecode*) razreda aplikacije i dodatne informacije
- ***Class loader subsystem*** – mehanizam za učitavanje tipova (razreda i sučelja) zadanih s **punim kvalificirajućim imenom** (npr. `java.awt.Rectangle`)
- ***Execution engine*** – mehanizam koji izvršava instrukcije međukoda

Unutarnji pogled na ustroj Javinog virtualnog stroja (JVM)

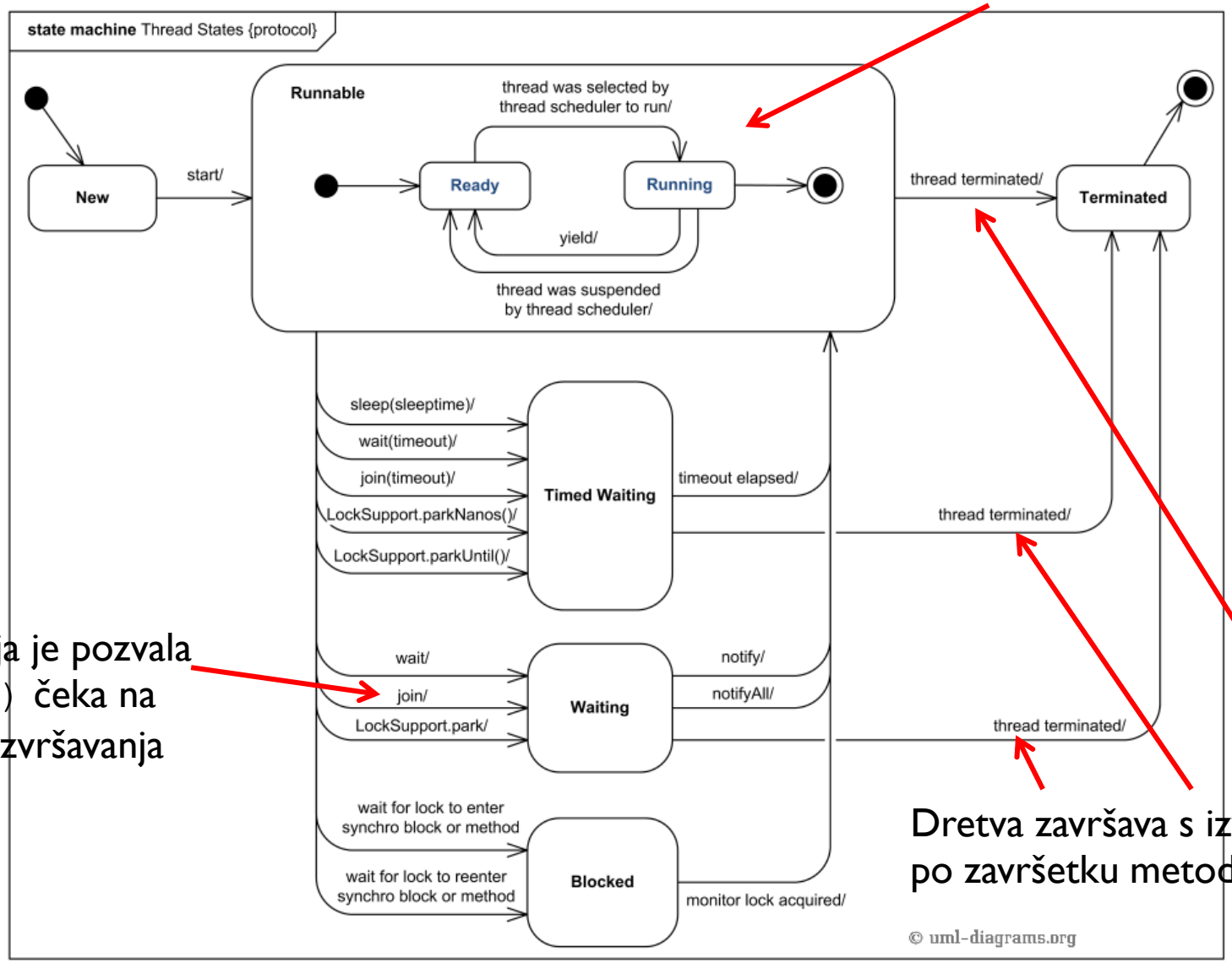
- ***Runtime data areas*** – memorijski prostori koje organizira JVM (ovisno od implementacije VM)
  - ***Method area*** – sadrži informacije o metodama, atributima, konstantama razreda i dr.
  - ***Heap*** – memorijski prostor za pohranjivanje objekata – tzv. **gomila**, rezerviran pri pokretanju VM, nad njim operira sakupljač smeća (*garbage collector*)
  - ***Java stacks i PC register*** – svaka dretva pri nastajanju dobije svoj stog i programsko brojilo (*PC register*). Stog je organiziran u okvire (*stack frames*), svaki okvir za jedan poziv metode – pamte se argumenti poziva, povratna vrijednost, lokalne varijable i međurezultati, samo je jedan okvir aktivan u jednom trenutku; programsko brojilo pokazuje adresu JVM instrukcije koja se izvodi
  - ***Native method stack*** – dretve koje izvršavaju nativne metode (metode ovisne o operacijskom sustavu), imaju za to rezerviran zasebni stog

Stanja dretvi u Javi

- **NEW** – dretva koja je instancirana (`Thread t = new Thread()`), ali nije još pokrenuta
- **RUNNABLE** – dretva koja je pokrenuta (`t.start()`), njezina metoda `run` se izvršava u JVM-u
- **BLOCKED** – dretva koja je blokirana čekajući na ulazak u kritični odsječak (monitor)
- **WAITING** – dretva čeka neodređeno dugo na drugu dretvu da izvrši neku akciju
- **TIMED_WAITING** – dretva čeka određeno vrijeme na drugu dretvu da izvrši neku akciju
- **TERMINATED** – dretva koja je završila izvođenje metode `run` se nalazi u ovom stanju

Stanja dretvi u Javi

Thread scheduler JVM-a upravlja redoslijedom izvođenja dretvi, pri čemu bitnu ulogu u odabiru redoslijeda izvođenja igra prioritet (*priority*) dretve



Dretva koja je pozvala `t.join()` čeka na završetak izvršavanja dretve `t`

Dretva završava s izvođenjem po završetku metode `run()`

Stanja dretvi u Javi

- Kritični odsječak u Javi može biti **metoda** ili **blok naredbi** ispred koje se nalazi ključna riječ `synchronized`, npr.

```
synchronized(obj) {
      while (<condition does not hold>) obj.wait();
      ...  // Perform action appropriate to condition
      notify();
}
```

- `synchronized` je Javin način za reći da postoji sinkronizacijski mehanizam – monitor za pristup kritičnom odsječku, samo jedna dretva mu može pristupiti u nekom trenutku
- Dretva koja poziva `wait()` ili `notify()` ili `notifyAll()` nad objektom mora se nalaziti u kritičnom odsječku (biti vlasnik monitora)
- Pozivom `wait()` dretva prestaje biti vlasnik monitora i čeka dok neka druga dretva ne pozove `notify()` nad tim objektom
- Pozivom `notify()` odabire se slučajno jedna dretva koja je čekala i koja se dalje natječe za izvođenje kritičnog odsječka kad trenutna dretva završi s njim
- Pozivom `notifyAll()` odabiru se sve dretve koje čekaju nad monitorom i onda se dalje natječu za izvođenje kritičnog odsječka

“Rješavanje” problema eksplozije broja stanja u JPF-u – smanjenje prostora stanja

- Prvi način: **provjera podudaranja stanja** (engl. *state matching*)
  - Na svakom grananju u kodu, provjerava se je li već istraženo isto stanje.
  - Ako se pronađe takvo stanje, onda se može sigurno obustaviti daljnja pretraga i vratiti se nazad na točku u kojoj još ima neistraženih putova
  - Dalje se od te ranije točke nastavlja pretraga prostora stanja.
  - Pri povratku na raniju točku, obnavlja se prijašnje stanje programa, što djeluje kao povratak debuggera unazad za *N* instrukcija!
  - Točno koje varijable programa ulaze u provjeru jednakosti stanja je nešto što se može korisnički odrediti

“Rješavanje” problema eksplozije broja stanja u JPF-u – smanjenje prostora stanja

- Drugi način: **djelomično smanjenje poretka** (engl. *partial order reduction*)
  - Broj različitih kombinacija pri raspoređivanju izvođenja dretvi je glavni razlog za veliki prostor stanja kod istovremenog izvođenja
  - Međutim, u većini praktičnih slučajeva nije potrebno razmatrati sve mogućnosti ispreplitanja izvođenja dretvi, budući da neki rasporedi izvršavanja programa dovode do istih stanja
  - Broj raspoređivanja dretvi smanjuje se tako da se grupiraju svi nizovi instrukcija u nekoj dretvi koji nemaju utjecaj izvan te dretve i to tako da se oni svedu na jedan prijelaz
  - Time se smanjuje do 70% prostora stanja programa
  - Djelomično smanjenje poretka ostvaruje se tako da se tijekom izvođenja prate instrukcije koje su bitne za raspoređivanje (npr. `synchronized` metode) ili one koje simuliraju nedeterminističku dodjelu vrijednosti varijabli te se prijelaz ostvaruje između takvih instrukcija.
  - Više na: https://github.com/javapathfinder/jpf-core/wiki/Partial-Order-Reduction

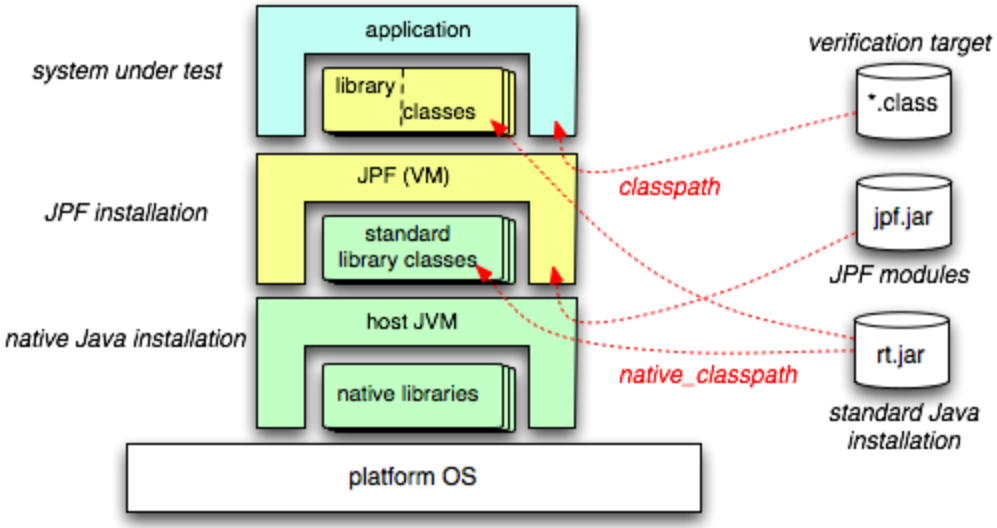
“Rješavanje” problema eksplozije broja stanja u JPF-u – smanjenje prostora stanja

- Treći način: **delegiranje izvođenja metoda JVM-u**
  - Neke instrukcije, i to one za koje se zna da ne utječu na verifikaciju (npr. `System.out.println()` delegiraju se za izvođenje host JVM-u umjesto da se izvršavaju unutar JPF-a

STRUKTURA JAVA PATHFINDERA

JPF i izvođenje programa

- JPF je ostvaren kao zasebni virtualni stroj (VM) koji se izvodi iznad instaliranog JVM-a za određeni OS (*host* JVM)
- JPF sam odlučuje koje dijelove našeg programa sam obrađuje, a koje delegira *host* JVM-u
- U pravilu, JPF značajno **usporava** izvođenje našeg programa (SUT) jer je to dodatni sloj iznad već postojećeg JVM-a



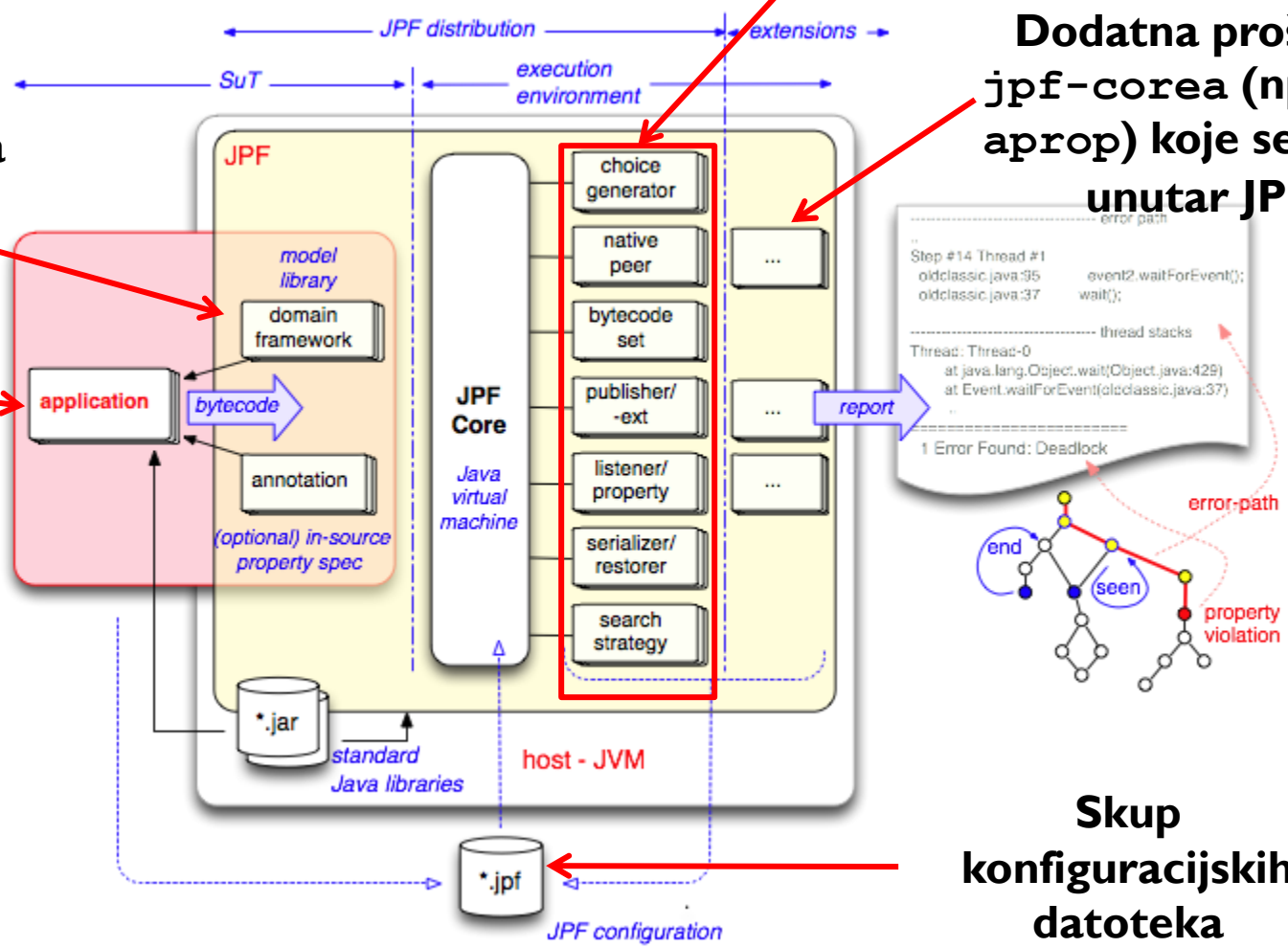
Struktura JPF-a

Standardni paketi `jpf-corea`

Dodatna proširenja `jpf-corea` (npr. `jpf-aprop`) koje se koriste unutar JPF-a

Knjižnice JPF-a koje koristi naša aplikacija

Naša aplikacija (datoteke *. class)



Skup konfiguracijskih datoteka

Struktura direktorija *src*

- **Main**
  - Paketi koji sadrže razrede ključne za rad `jpf-corea` (npr. algoritmi pretraživanja prostora stanja, izvršavanje VM, slušači i dr.). Oni se izvode na ***host* JVM-u** (kao što je specificirano u JPF `native_classpath`)
- **Peers**
  - *Native peer classes* – Paketi s ostalim razredima koji sadrže implementaciju metoda koje se izvode umjesto pravih nativnih metoda. Izvode se na ***host* JVM** (isto specificirano u JPF `native_classpath`)
- **Classes**
  - *Model classes* – Razredi koji se izvršavaju **izravno na VM-u od JPF-a**, a koje može importirati i koristiti naša aplikacija, služe kao knjižnice JPF-a za našu aplikaciju (specificirani su u JPF `classpath`)
- **Annotations**
  - Sadrži Javine anotacije (koje počinju sa znakom "@") koje treba obraditi JPF. Naša aplikacija ih može koristiti ako se provjeravaju neka svojstva (npr. @JPFConfig, @FilterField)
- **Examples**
  - Sadrži primjere SUT-ova i njima odgovarajućih konfiguracijskih datoteka sa specificiranim svojstvima
- **Tests**
  - Sadrži tipične ispitne slučajeve (testove) za veći broj standardnih Java razreda kao i za ispitivanje rada samog JPF-a

Struktura direktorija `build`

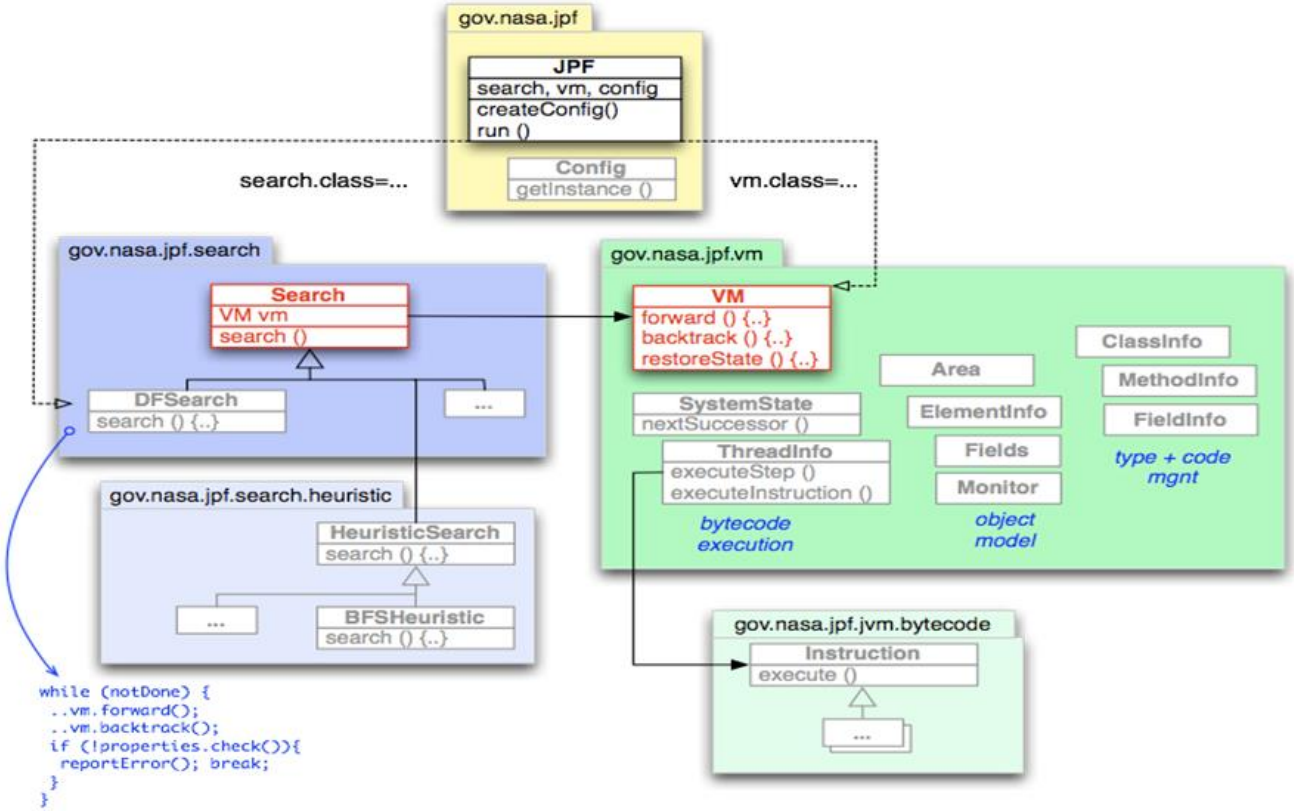
- Slična kao i struktura direktorija `src` (podjela na šest poddirektorija)
- Direktorij `build` stvara se izgradnjom (engl. *build*) – kompajliranjem svih razreda (`*.java`) iz direktorija `src`
- U tu svrhu koristi se sustav Ant za izgradnju aplikacije – postoji već unaprijed napisana skripta za izgradnju JPF-a koja se samo pokrene, najbolje iz razvojnog okruženja (npr. NetBeans) ili iz Anta
- Najznačajnija za pokretanje JPF-a je (Java-izvršna) datoteka `RunJPF.jar` koja je izgrađena u korijenu direktorija `build`

Ostali značajniji direktoriji

- Direktorij `lib`
  - Tu se stavljaju dodatne knjižnice koje se mogu koristiti uz `jpf-core` (npr. JUnit za ispitivanje)
- Direktorij `doc`
  - Dokumentacija `jpf-corea` u tekstualnom obliku (*.md – markdown *readme* datoteke)

Poddirektorij `main` – najvažniji dio `jpf-core`

- **VM** (`gov.nasa.jpf.vm.VM`) i **Search** (`gov.nasa.jpf.search.Search`) su najbitniji
- `VM` proizvodi stanja programa i pomiče se korak unaprijed ili unatrag
- `Search` je upravljački program nad `VM`-om koji omogućuje pretraživanje stanja programa korištenjem raznih strategija (DFSearch, RandomSearch…) i heuristika, ovisno o specifikaciji



SPECIFIKACIJA I PROVJERA SVOJSTAVA PROGRAMA

Specifikacija i provjera svojstava programa

- Specifikacija svojstava koje treba provjeriti u JPF-u nije jednostavna kao u drugim sustavima (npr. NuSMV)
- Svojstva se ne navode u vremenskoj logici, već se navodi koja svojstva programa treba provjeriti (npr. Nonnull, Deadlock...)
- JPF je visoko konfigurabilan, čime je omogućena velika proširivost sustava, no nedostatak je velika složenost konfiguracije
- Različiti dijelovi sustava mogu imati različite vlastite parametre (vrsta pretrage, slušači, skupovi instrukcija...)
- Time je onemogućeno postojanje konfiguracijskog objekta koji bi sadržavao konkretna unaprijed definirana polja za parametre

Specifikacija i provjera svojstava programa

- Konfiguracijski objekt je fleksibilan:
  - zasnovan je na nizovima znakova (String)
  - po volji je proširiv
  - prenosi se odozgo prema dolje u hijerarhijskom procesu tako da svaka komponenta izvuče samo svoje potrebne parametre
- Konfiguracijski objekt ostvaren je razredom `Config` (u paketu `gov.nasa.jpf`)
- On obavještava daljnje razrede pri promjeni bilo kojih parametara, a daljnji razredi provjeravaju parametre i ako utvrde da su zaduženi za njihovu obradu, onda ih obrađuju

Specifikacija i provjera svojstava programa

- Konfiguracija sustava se ostvaruje na četiri razine:
  - Cjelokupne instalacije na računalu
  - Projektne instalacije (za svaku komponentu JPF-a koja je instalirana)
  - Korisničke aplikacije (SUT)
  - Komandno-linijski argumenti (prema potrebi)
- **Niža razina uvijek može nadjačati svojstva (konfiguraciju) više razine!**

Podešavanje cjelokupne instalacije

- Datoteka s parametrima za cjelokupnu JPF instalaciju naziva se `site.properties` i treba se nalaziti na lokaciji `<user.home>/.jpf/site.properties`
- To je uobičajena datoteka vrste ***Java properties,*** što znači da je većina svojstava definirana po principu:

  <ključ> = <vrijednost>
- Ova datoteka govori JPF-u prilikom pokretanja gdje da traži projekte u okviru JPF-a koji su instalirani na računalu, kako bi podesio njihove *classpathove* bez potrebe za time da ih korisnik sam svaki put mora iznova upisivati
- Datoteku `site.properties` potrebno je ručno napisati prilikom prve instalacije JPF-a, jer u suprotnom JPF neće raditi

Podešavanje cjelokupne instalacije

- Primjer datoteke `site.properties`:

```
# JPF site configuration

jpf.home = ${user.home}/Documents/NetBeansProjects

# can only expand system properties
jpf-core = ${jpf.home}/jpf-core
extensions=${jpf-core}

# annotation properties extension
jpf-aprop = ${jpf.home}/jpf-aprop
extensions+=,${jpf-aprop}

# numeric extension
jpf-numeric = ${jpf.home}/jpf-numeric
extensions+=,${jpf-numeric}
```

Definiramo lokaciju za `jpf.home`

Lokacija instalacije `jpf.core`

`extensions` definira sva proširenja (sve instalirane projekte). U novijim verzijama JPF-a `extensions` nije nužno eksplicitno navoditi u `site.properties`.

Lokacije ostalih projekata koje treba uzeti u obzir

Napomene: ostali projekti se mogu navesti u `site.properties` čak i ako nisu instalirani na računalu. JPF će ih u tom slučaju ignorirati. Dodatni projekt našeg SUT-a ne treba se navoditi u datoteci `site.properties`, ona služi samo za JPF!

Podešavanje projektne instalacije

- Svaki projekt u JPF-u (uključujući `jpf-core` i sva proširenja) sadržava u svojem **korijenskom** direktoriju datoteku `jpf.properties`
- U toj datoteci navode se svi parametri karakteristični za taj projekt u cjelini
- Tipično se tu navode *classpathovi* do svih JPF aplikacija koje projekt sadrži, svojstva koja se po *defaultu* provjeravaju kao i ostali parametri svojstveni tom projektu
- Svaki korisnički projekt koji će biti SUT također može (ali ne mora) u svojem korijenskom direktoriju imati datoteku `jpf.properties`
- Datoteke `jpf.properties` izvršavaju se onim redoslijedom koji je određen u datoteci `site.properties`, uz iznimku ako se JPF pokreće unutar određenog projekta, u kojem slučaju se najprije učitava `jpf.properties` za taj projekt
- Bitno je da su `jpf.properties` i `site.properties` međusobno konzistentne, što znači da nazivi projekata odgovaraju (npr. "`jpf-aprop`" mora biti isti u datotekama `site.properties` i `jpf.properties`)

Podešavanja projektne instalacije

- Primjer datoteke `jpf.properties`:

```
jpf-aprop = ${config_path}

#--- path specifications
jpf-aprop.native_classpath = build/jpf-aprop.jar;lib/antlr-runtime-3.1.3.jar
jpf-aprop.classpath = build/examples
jpf-aprop.test_classpath = build/tests
jpf-aprop.sourcepath = src/examples

#--- other project specific settings
listener=${listener.autoload},javax.annotation.Nonnull,...
listener.javax.annotation.Nonnull=gov.nasa.jpf.aprop.listener.NonnullChecker
```

Prvo svojstvo uvijek definira naziv projekta, {config_path} se uvijek proširuje nazivom direktorija gdje se nalazi `jpf.properties`

Put do razreda koji će izvršavati programe, a koji mora biti vidljiv *host* JVM-u

Put do direktorija koji sadrže primjere (SUT)

Put do direktorija koji sadrže uobičajene testove

Put do direktorija koji sadrže izvorni kod, što se koristi ako JPF treba generirati trag programa

Postavljanje dodatnih svojstava specifičnih za projekt `jpf-aprop`, konkretno u ovom slučaju učitava se slušač (*listener*) koji provjerava anotacijsko svojstvo @Nonnull. Sada više nije potrebno u SUT konfiguracijskim datotekama navoditi tog slušača!

Podešavanje svojstava SUT-a

- Put do direktorija gdje se nalaze datoteke `*.class` SUT-a definira se najčešće u datoteci `jpf.properties` (svojstvo `.classpath`)
- Podešavanje svojstava **specifičnih za određeni SUT** navodi se u datoteci `*.jpf`, koja se može nalaziti bilo gdje, no obično u istom direktoriju gdje je i izvorni kod samog razreda koji se provjerava
- Uobičajeno je dati naziv toj datoteci takav da odgovara nazivu razreda aplikacije koja se ispituje, npr. `oldclassic.jpf`, ali to nije nužan uvjet
- Kako bi se pokrenuo JPF, u datoteci `*.jpf` **nužno** je definirati **koji glavni razred aplikacije (razred koji sadrži metodu `main`) je potrebno izvršiti** (ključ `target`)
- Mogu se navesti argumenti koje treba primiti `static` metoda `main` tog glavnog razreda (ključ `target_args`)
- Osim toga, navodi se niz svojstava koja specificiraju kako želimo provjeriti našu aplikaciju (slušači, način pretraživanja, način i poredak izvještavanja...)

Podešavanje svojstava SUT-a

- Primjer datoteke (skraćeno) `RobotManager.jpf`

```
#--- dependencies on other JPF modules
@using = jpf-awt
@using = jpf-shell
```

"@using=<ime_projekta>" je naredba JPF-u da učita `jpf.properties` datoteku navedenog projekta (koji mora biti definiran u datoteci `site.properties`). Ovako se uklanja potreba za navođenjem `extensions` u datoteci `site.properties`

```
#--- what JPF should run
target = RobotManager
```

Definiranje naziva ciljnog glavnog razreda aplikacije koji se pokreće da bi izvršila provjera modela

Dodatni slušač, uz one navedene u `jpf.properties` datotekama koje se uzima u obzir. Ovaj konkretni slušač nalazi se u `jpf-core:gov.nasa.jpf.listener.OverlappingMethodAnalyzer`

```
#--- other stuff that defines how to run JPF
listener+=,.listener.OverlappingMethodAnalyzer
cg.enumerate_random=true
```

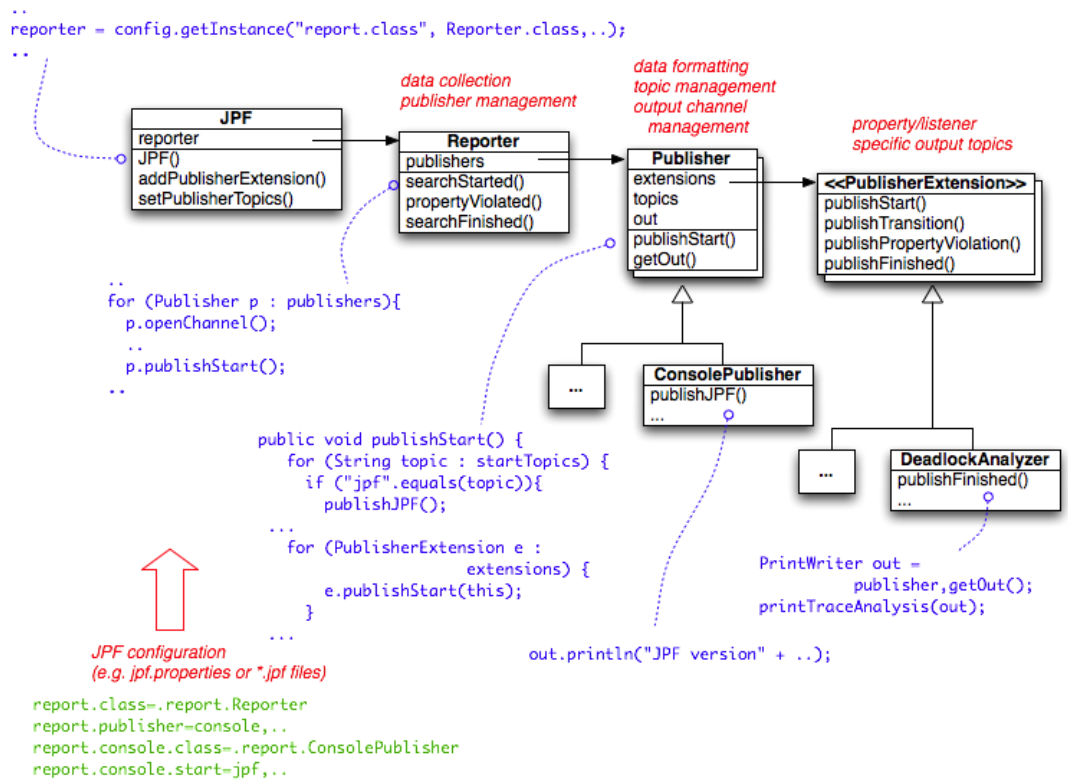
Specifikacija svojstva da se pobroje sve mogućnosti prilikom slučajnog izbora koji se pojavljuje unutar programa (npr. `Random.nextInt(2);` će dati dvije mogućnosti: 0 ili 1). Ključ `cg.enumerate_random` ispituje se *u Native peeru* `JPF_java_util_Random` projekta `jpf-core`

Podešavanje svojstava korištenjem komandne linije

- Komandnom linijom JPF-a moguće je nadjačati sve gornje razine hijerarhije konfiguracije
- U tom slučaju, najčešće se dodaju svojstva korištenjem notacije `<ključ>+=<vrijednost>`, iako su moguće i drugačije notacije, vidjeti:
- https://github.com/javapathfinder/jpf-core/wiki/Configuring-JPF (dio *Special Property syntax*)
- Najfleksibilnije, ali uglavnom nepotrebno rješenje osim u slučaju *debuggiranja* samog JPF-a
- Mora se točno znati što se radi ☺

Izvještaji

- Za generiranje konačnog izvještaja (engl. *report*) o provjeri modela zaduženi su:
  - Razred izvjestitelj - Reporter
  - Razredi izdavači - Publisher, čiji tipovi ovise o specifikaciji vrste izvještavanja
  - Razredi koji proširuju izdavače - PublisherExtensions – specifični izdavači za određena svojstva



Izvještaji

- Izvjestitelj upravlja i obavještava izdavače kada se dođe do pojedine faze izvještavanja (*start, property_violation, finish*)
- Faza *property_violation* ima određene teme izvještavanja
  - ***error*** – prikazuje tip i detalje kršenja svojstva koje je pronađeno
  - ***trace*** – prikazuje trag programa koji dovodi do kršenja svojstva
  - ***snapshot*** – daje listu stanja svake dretve u trenutku kršenja svojstva
  - ***output*** – prikazuje izlaz iz programa za taj trag
- Faza *finish* ima po *defaultu* postavljene teme izvještavanja
  - ***results*** – prikazuje je li došlo do kršenja svojstava i daje kratku listu pogrešaka
  - ***statistics*** – prikazuje ukupnu statistiku izvođenja JPF-a na SUT-u
- Izdavači proizvode izlaz iz sustava u željenom obliku (npr. tekst, XML). Uobičajeni izdavač je `ConsolePublisher` (`report.console.*` za izvještavanje u konzoli u obliku teksta)
- Za svaki simbolički naziv izdavača i fazu izvještavanja potrebno je specificirati kojim redom se teme prikazuju (u datoteci `*.jpf`):

Primjer:

```
report.console.property_violation=error,trace,snapshot
```

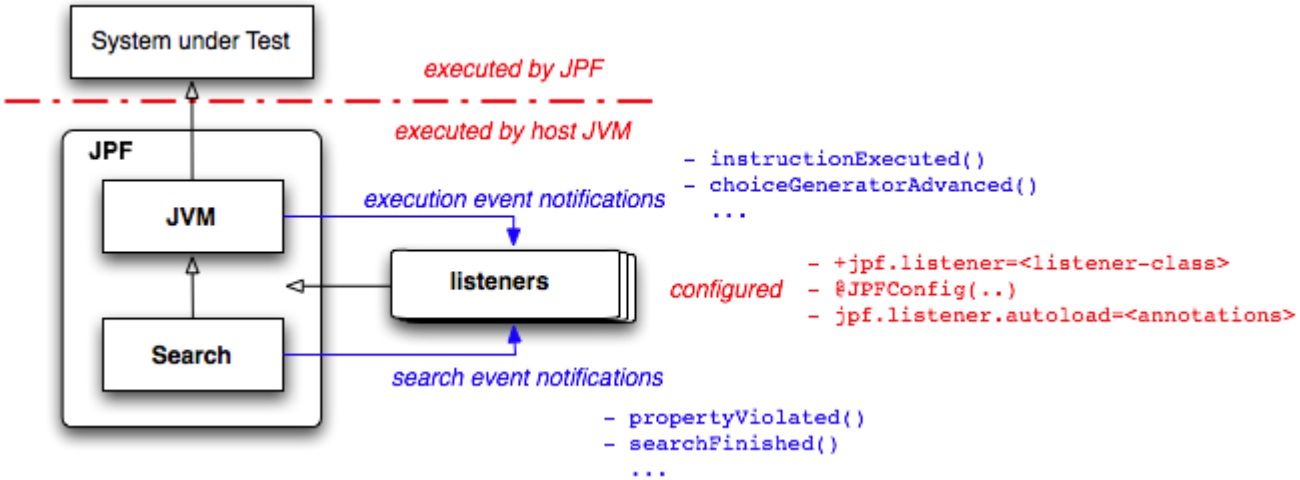
NAPREDNE TEME I PROŠIRENJA

Napredne teme i proširenja JPF-a

- Slušači (engl. *Listeners*)
- Generatori izbora (engl. *Choice generators*)
- Provjera anotacijskih svojstvava – `jpf-aprop`

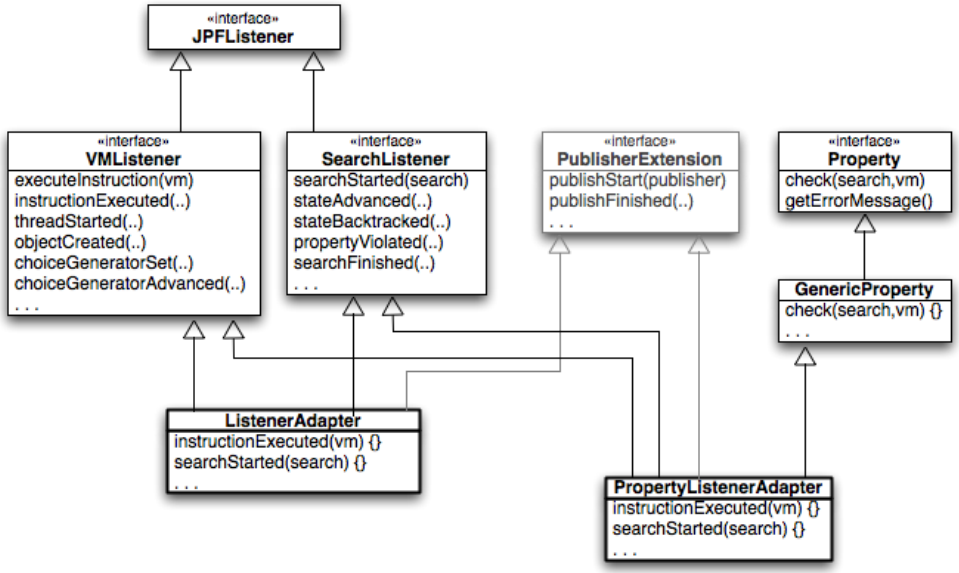
Slušači

- Najvažniji mehanizam proširenja u JPF-u
- Pružaju način za opažanjem, interakcijom i proširenjem izvođenja JPF-a
- Slušači su *Observeri* koji reagiraju na određeni događaj prilikom pretrage (`SearchListener`) ili rada JVM-a od JPF-a (`VMListener`)
- Konfiguriraju se navođenjem svojstva `listener` u `jpf.properties` ako vrijede za cijeli projekt, u datoteci `*.jpf` (najčešće), ili kroz komandnu liniju
- JPF također pokreće odgovarajućeg pridruženog slušača kad naiđe na anotaciju @JPFConfig s navedenim slušačem u datoteci s izvornim kodom



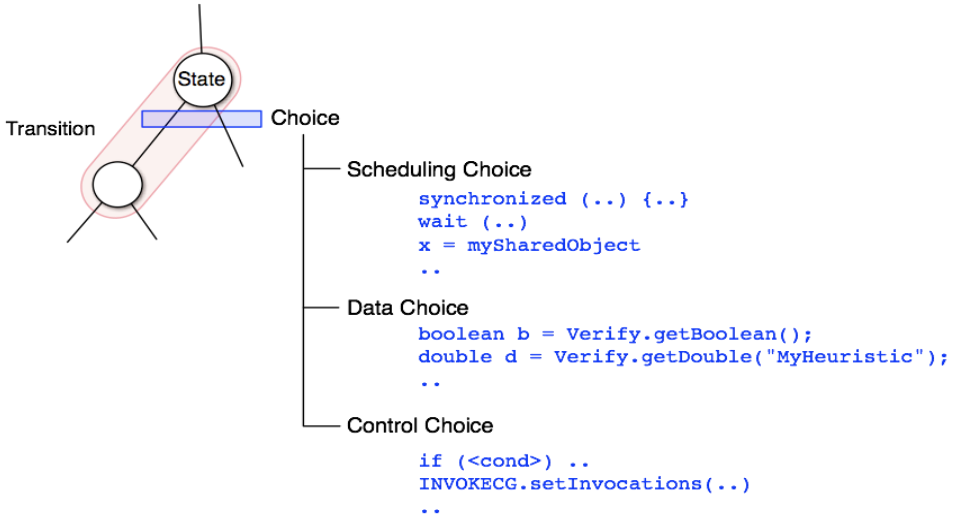
Slušači

- Konkretni slušači obično nasljeđuju jedan razred *Adapter* koji ima prazne implementacije odgovarajućih metoda sučelja `VMListener`, `SearchListener` i drugih (npr. `NullTracker` **`extends`** `ListenerAdapter`) i zatim nadjačavaju metode dojave za koje su zainteresirani
- `ListenerAdapter` služi za skupljanje informacija o izvođenju JPF-a pa će ga koristiti slušači npr. `CoverageAnalyzer`, `DeadlockAnalyzer` i `NullTracker`
- `PropertyListenerAdapter` se koristi kada slušač implementira neko svojstvo programa povezano s pretragom (npr. `PreciseRaceDetector`, `NoStateCycles`)



Generatori izbora

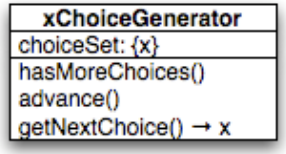
- Provjera modela ima zadatak doći do zanimljivih stanja programa uz ograničene resurse koji su na raspolaganju
- *ChoiceGenerators* je mehanizam JPF-a da sustavno istražuje prostor stanja kako bi došao do rješenja
- Veći broj postojećih mogućnosti izbora: **raspoređivanje dretvi, vrijednosti podataka, kontrolni tok programa**
- Mehanizam odabira odgovarajućeg generatora izbora i parametara heuristike odvojen je od samog programa – navodi se u konfiguracijskoj datoteci kao svojstvo odgovarajućeg imena zajedno s ostalim svojstvima



Generatori izbora

- Kod provjere vrijednosti podataka, kako bi se smanjio broj mogućnosti dodjele vrijednosti pri provjeri `int`, `double` i drugih tipova varijabli, uvode se heuristike pretraživanja

`Verify.getBoolean()`	C = { true, false }	✓	
`Verify.getInt(0,4)`	C = { 0, 1, 2, 3, 4 }	?	potentially large sets with lots of uninteresting values
`Verify.getDouble(1.0,1.5)`	C = { ∞ }	??	no finite value set without heuristics

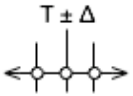


Choice Generators — JPF internal object to store and enumerate a set of choices

+

Configurable Heuristic Choice Models — configurable classes to create ChoiceGenerator instances

e.g. "Threshold" heuristic



C = { T-Δ, T, T+ Δ }

application code (test driver)

```
..
double v = Verify.getDouble("velocity");
..
```

configuration (e.g. mode property file)

```
velocity.class = gov.nasa.jpf.jvm.choice.DoubleThresholdGenerator
velocity.threshold = 13250
velocity.delta = 500
```

Generatori izbora

- Oprez: korištenjem heuristika odstupa se od temeljnog (ali idealističkog) zahtjeva kod provjere modela: da se ispitaju svi putovi kroz program, u ovom slučaju ispitaju se samo oni zanimljivi putovi (zanimljivost je subjektivan kriterij!)
- Interno, u JPF-u, generatori izbora omogućuju odabir pri postupku raspoređivanja dretvi našeg programa koje se u JPF-u izvodi kao *bytecode* (`ChoiceGenerator` i `ThreadChoiceGenerator`),
- Detaljnije: https://github.com/javapathfinder/jpf-core/wiki/ChoiceGenerators

Provjera anotacijskih svojstava

- Projekt `jpf-aprop` predstavlja proširenje `jpf-corea` s namjenom provjere modela za specificirana anotacijska svojstva programa
- Ako se u nekom trenutku žele provjeriti anotacijska svojstva, odgovarajući slušači će se navesti u `*.jpf` konfiguracijskoj datoteci te će se program moći prevesti i provjeriti
- U idealnom slučaju, ove anotacije su korisne i za dokumentiranje programa, a može ih se obraditi i s nekim alatima za statičku analizu koda
- Izvedbeno, prije prevođenja SUT-a, potrebno je napraviti *import* odgovarajućih razreda koji su zaduženi za dotične anotacije te uključiti knjižnicu `jpf-aprop-annotations.jar` u kojoj se nalazi popis razreda s anotacijama.

Provjera anotacijskih svojstava

- Struktura projekta `jpf-aprop` je slična projektu `jpf-core`, samo je `jpf-aprop` bitno manji i ima jednostavniju datoteku `jpf.properties`
- Anotacijska svojstva pokrivaju sljedeće zadatke:
  - Nedozvoljeno dodjeljivanje `null` vrijednosti (@Nonnull)
  - Ugovore (@Requires, @Ensures, @Invariant)
  - Obilježja sigurnosti pristupa polju razreda kod dretvi (@GuardedBy)
  - Promijenjivost objekta (@Const)
  - i drugo...

INSTALACIJA

Instalacija JPF-a

1. Instalacija Jave i NetBeans IDE-a
2. Kloniranje repozitorija (`jpf-core`) s GitHuba
3. Izgradnja projekta `jpf-core` korištenjem Ant-skripte `build.xml`
4. Izrada datoteke `site.properties`
5. Instalacija *plugina* za NetBeans za provjeru modela ("Verify...")
6. Spremni! ☺

O 2. domaćoj zadaći

- Instalacija JPF-a

- Sama zadaća je podijeljena u 4 dijela:

1. Upoznavanje s projektom `jpf-core` i pokretanje provjere modela jednostavnih primjera programa.
2. Uvode se dodatni razredi slušači koji nadograđuju osnovnu funkcionalnost projekta `jpf-core`.
3. Pokriveno je izvođenje provjere modela nad primjerima iz dodatnog projekta `jpf-aprop`
4. Verifikacija korisničkog projekta, uključivanje korištenja projekata `jpf-core` i `jpf-aprop` te provjera modela zadanog programa uz stalne izmjene.

O čemu nismo detaljno govorili

- Model Java Interface (MJI)
  - Mehanizam preusmjeravanja izvođenja metoda na JPF-u ili od JPF-a na JVM
  - https://github.com/javapathfinder/jpf-core/wiki/Model-Java-Interface
- *Bytecode factories*
  - Kako JPF interno procesira instrukcije *bytecodea* programa
  - https://github.com/javapathfinder/jpf-core/wiki/Bytecode-Factories
- *Logging*
  - Bilježenje pogrešaka u ovisnosti o ozbiljnosti (*severity*) pogreške u dnevnik (*log*)
  - https://github.com/javapathfinder/jpf-core/wiki/Logging-system
- *Symbolic Pathfinder*
  - Izvršavanje programa korištenjem simboličkih (ograničenih numeričkih) vrijednosti ulaznih varijabli na temelju analize koda – koristi se uglavnom za automatizaciju procesa ispitivanja
  - https://github.com/SymbolicPathFinder/jpf-symbc/wiki
- Velik broj ostalih proširenja:
  - https://github.com/javapathfinder/jpf-core/wiki/Projects

Zaključak

- JPF je napredan i proširiv radni okvir za provjeru modela programa pisanog u Javi
- Nadograđuje Javin VM svojim vlastitim VM što mu omogućuje nesmetano kretanje kroz stanja programa
- Koristi više tehnika za smanjenje prostora pretraživanja
- Relativno složena konfiguracija projekata koja omogućava proširivost, ali i komplicira uporabu